Finite difference methods for wave motion

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 5, 2014

This is still a **preliminary version**.

Contents

Simulation of waves on a string **3**
1.1 Discretizing the domain . . . 4
1.2 The discrete solution . . . 5
1.3 Fulfilling the equation at the mesh points . . . 6
1.4 Replacing derivatives by finite differences . . . 6
1.5 Formulating a recursive algorithm . . . 7
1.6 Sketch of an implementation . . . 8

Verification **9**
2.1 A slightly generalized model problem . . . 9
2.2 Using an analytical solution of physical significance . . . 10
2.3 Manufactured solution . . . 11
2.4 Constructing an exact solution of the discrete equations . . . 12

Implementation **14**
3.1 Making a solver function . . . 14
3.2 Verification: exact quadratic solution . . . 15
3.3 Visualization: animating the solution . . . 16
3.4 Running a case . . . 19
3.5 The benefits of scaling . . . 20

Vectorization **21**
4.1 Operations on slices of arrays . . . 21
4.2 Finite difference schemes expressed as slices . . . 23
4.3 Verification . . . 24
4.4 Efficiency measurements . . . 25
4.5 Remark on the updating of arrays . . . 25

5 Exercises

6 Generalization: reflecting boundaries
6.1 Neumann boundary condition . . .
6.2 Discretization of derivatives at the boundary . . .
6.3 Implementation of Neumann conditions . . .
6.4 Index set notation . . .
6.5 Alternative implementation via ghost cells . . .

7 Generalization: variable wave velocity
7.1 The model PDE with a variable coefficient . . .
7.2 Discretizing the variable coefficient . . .
7.3 Computing the coefficient between mesh points . . .
7.4 How a variable coefficient affects the stability . . .
7.5 Neumann condition and a variable coefficient . . .
7.6 Implementation of variable coefficients . . .
7.7 A more general model PDE with variable coefficients . . .
7.8 Generalization: damping . . .

8 Building a general 1D wave equation solver
8.1 User action function as a class . . .
8.2 Pulse propagation in two media . . .

9 Exercises

10 Analysis of the difference equations
10.1 Properties of the solution of the wave equation . . .
10.2 More precise definition of Fourier representations . . .
10.3 Stability . . .
10.4 Numerical dispersion relation . . .
10.5 Extending the analysis to 2D and 3D . . .

11 Finite difference methods for 2D and 3D wave equations
11.1 Multi-dimensional wave equations . . .
11.2 Mesh . . .
11.3 Discretization . . .

12 Implementation
12.1 Scalar computations . . .
12.2 Vectorized computations . . .
12.3 Verification . . .

13 Migrating loops to Cython
13.1 Declaring variables and annotating the code . . .
13.2 Visual inspection of the C translation . . .
13.3 Building the extension module . . .
13.4 Calling the Cython function from Python . . .

4 Migrating loops to Fortran **77**
14.1 The Fortran subroutine 77
14.2 Building the Fortran module with f2py 79
14.3 How to avoid array copying 80

5 Migrating loops to C via Cython **82**
15.1 Translating index pairs to single indices 82
15.2 The complete C code 83
15.3 The Cython interface file 83
15.4 Building the extension module 84

6 Migrating loops to C via f2py **85**
16.1 Migrating loops to C++ via f2py 86

7 Using classes to implement a simulator **87**

8 Exercises **87**

9 Applications of wave equations **88**
19.1 Waves on a string 89
19.2 Waves on a membrane 92
19.3 Elastic waves in a rod 92
19.4 The acoustic model for seismic waves 92
19.5 Sound waves in liquids and gases 94
19.6 Spherical waves 95
19.7 The linear shallow water equations 96
19.8 Waves in blood vessels 98
19.9 Electromagnetic waves 100

0 Exercises **101**

A very wide range of physical processes lead to wave motion, where signals re propagated through a medium in space and time, normally with little or o permanent movement of the medium itself. The shape of the signals may ndergo changes as they travel through matter, but usually not so much that ıe signals cannot be recognized at some later point in space and time. Many ypes of wave motion can be described by the equation $u_{tt} = \nabla \cdot (c^2 \nabla u) + f$, hich we will solve in the forthcoming text by finite difference methods.

Simulation of waves on a string

Ve begin our study of wave equations by simulating one-dimensional waves on string, say on a guitar or violin string. Let the string in the deformed state ɔincide with the interval $[0, L]$ on the x axis, and let $u(x, t)$ be the displacement

at time t in the y direction of a point initially at x. The displacement f u is governed by the mathematical model

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= c^2 \frac{\partial^2 u}{\partial x^2}, && x \in (0, L),\ t \in (0, T] \\
u(x, 0) &= I(x), && x \in [0, L] \\
\frac{\partial}{\partial t} u(x, 0) &= 0, && x \in [0, L] \\
u(0, t) &= 0, && t \in (0, T] \\
u(L, t) &= 0, && t \in (0, T]
\end{aligned}$$

The constant c and the function $I(x)$ must be prescribed.

Equation (1) is known as the one-dimensional *wave equation*. Since tl contains a second-order derivative in time, we need *two initial conditio* (2) specifying the initial shape of the string, $I(x)$, and (3) reflecting t initial velocity of the string is zero. In addition, PDEs need *boundary co* here (4) and (5), specifying that the string is fixed at the ends, i.e., t displacement u is zero.

The solution $u(x, t)$ varies in space and time and describes waves t moving with velocity c to the left and right.

Sometimes we will use a more compact notation for the partial deriva save space:

$$u_t = \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2},$$

and similar expressions for derivatives with respect to other variables. T wave equation can be written compactly as $u_{tt} = c^2 u_{xx}$.

The PDE problem (1)-(5) will now be discretized in space and tir finite difference method.

1.1 Discretizing the domain

The temporal domain $[0, T]$ is represented by a finite number of mesh p

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t - 1} < t_{N_t} = T\,.$$

Similarly, the spatial domain $[0, L]$ is replaced by a set of mesh points

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x - 1} < x_{N_x} = L\,.$$

One may view the mesh as two-dimensional in the x, t plane, consisting c (x_i, t_n), with $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$.

Uniform meshes. For uniformly distributed mesh points we can introduce ıe constant mesh spacings Δt and Δx. We have that

$$x_i = i\Delta x,\ i = 0, \ldots, N_x, \quad t_i = n\Delta t,\ n = 0, \ldots, N_t\,. \tag{9}$$

Ve also have that $\Delta x = x_i - x_{i-1}$, $i = 1, \ldots, N_x$, and $\Delta t = t_n - t_{n-1}$, $n = \ldots, N_t$. Figure 1 displays a mesh in the x, t plane with $N_t = 5$, $N_x = 5$, and onstant mesh spacings.

.2 The discrete solution

he solution $u(x,t)$ is sought at the mesh points. We introduce the mesh unction u_i^n, which approximates the exact solution at the mesh point (x_i, t_n) or $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$. Using the finite difference method, we shall evelop algebraic equations for computing the mesh function. The circles in igure 1 illustrate neighboring mesh points where values of u_i^n are connected hrough an algebraic equation. In this particular case, u_2^1, u_1^2, u_2^2, u_3^2, and u_2^3 are onnected in an algebraic equation associated with the center point $(2, 2)$. The erm *stencil* is often used about the algebraic equation at a mesh point, and the eometry of a typical stencil is illustrated in Figure 1. One also often refers to ıe algebraic equations as *discrete equations*, *(finite) difference equations* or a *nite difference scheme*.

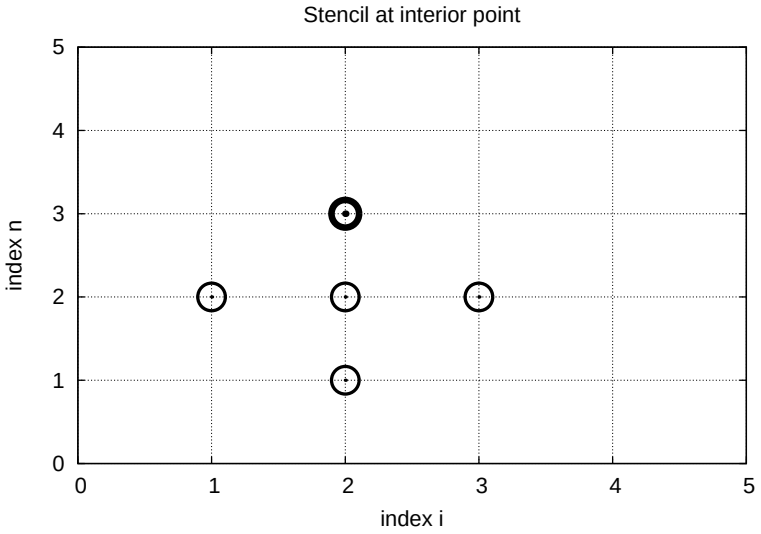


Figure 1: Mesh in space and time for a 1D wave equation.

1.3 Fulfilling the equation at the mesh points

For a numerical solution by the finite difference method, we relax the cc that (1) holds at all points in the space-time domain $(0, L) \times (0, T]$ requirement that the PDE is fulfilled at the *interior* mesh points:

$$\frac{\partial^2}{\partial t^2}u(x_i, t_n) = c^2\frac{\partial^2}{\partial x^2}u(x_i, t_n),$$

for $i = 1, \ldots, N_x - 1$ and $n = 1, \ldots, N_t - 1$. For $n = 0$ we have th conditions $u = I(x)$ and $u_t = 0$, and at the boundaries $i = 0, N_x$ we h boundary condition $u = 0$.

1.4 Replacing derivatives by finite differences

The second-order derivatives can be replaced by central differences. T widely used difference approximation of the second-order derivative is

$$\frac{\partial^2}{\partial t^2}u(x_i, t_n) \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}\,.$$

It is convenient to introduce the finite difference operator notation

$$[D_t D_t u]_i^n = \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}\,.$$

A similar approximation of the second-order derivative in the x directic

$$\frac{\partial^2}{\partial x^2}u(x_i, t_n) \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} = [D_x D_x u]_i^n\,.$$

Algebraic version of the PDE. We can now replace the derivative and get

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2},$$

or written more compactly using the operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n\,.$$

Algebraic version of the initial conditions. We also need to rep derivative in the initial condition (3) by a finite difference approxima centered difference of the type

$$\frac{\partial}{\partial t}u(x_i, t_n) \approx \frac{u_i^1 - u_i^{-1}}{2\Delta t} = [D_{2t}u]_i^0,$$

seems appropriate. In operator notation the initial condition is written

$$[D_{2t}u]_i^n = 0, \quad n = 0\,.$$

/riting out this equation and ordering the terms give

$$u_i^{n-1} = u_i^{n+1}, \quad i = 0, \ldots, N_x, \ n = 0\,. \tag{13}$$

he other initial condition can be computed by

$$u_i^0 = I(x_i), \quad i = 0, \ldots, N_x\,.$$

.5 Formulating a recursive algorithm

/e assume that u_i^n and u_i^{n-1} are already computed for $i = 0, \ldots, N_x$. The only nknown quantity in (11) is therefore u_i^{n+1}, which we can solve for:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \tag{14}$$

here we have introduced the parameter

$$C = c\frac{\Delta t}{\Delta x}, \tag{15}$$

nown as the *Courant number*.

C is the key parameter in the discrete diffusion equation.

We see that the discrete version of the PDE features only one parameter, C, which is therefore the key parameter that governs the quality of the numerical solution (see Section 10 for details). Both the primary physical parameter c and the numerical parameters Δx and Δt are lumped together in C. Note that C is a dimensionless parameter.

Given that u_i^{n-1} and u_i^n are computed for $i = 0, \ldots, N_x$, we find new values t the next time level by applying the formula (14) for $i = 1, \ldots, N_x - 1$. Figure 1 lustrates the points that are used to compute u_2^3. For the boundary points, $= 0$ and $i = N_x$, we apply the boundary conditions $u_i^{n+1} = 0$.

A problem with (14) arises when $n = 0$ since the formula for u_i^1 involves u_i^{-1}, hich is an undefined quantity outside the time mesh (and the time domain). owever, we can use the initial condition (13) in combination with (14) when $= 0$ to arrive at a special formula for u_i^1:

$$u_i^1 = u_i^0 - \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)\,. \tag{16}$$

igure 2 illustrates how (16) connects four instead of five points: u_2^1, u_1^0, u_2^0, and $_3^0$.

We can now summarize the computational algorithm:

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$

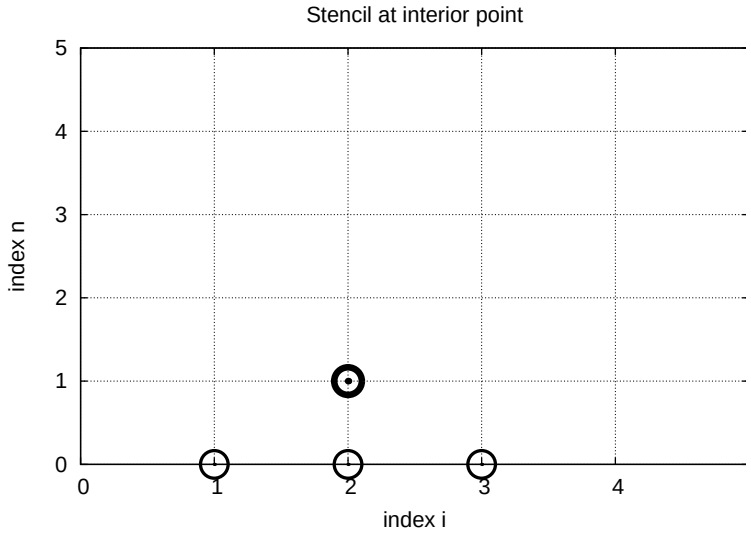


Figure 2: Modified stencil for the first time step.

2. Compute u_i^1 by (16) and set $u_i^1 = 0$ for the boundary points $i =$ $i = N_x$, for $n = 1, 2, \ldots, N - 1$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
   (a) apply (14) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
   (b) set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

The algorithm essentially consists of moving a finite difference stencil all the mesh points, which is illustrated by an animation in a web pa movie file[2].

1.6 Sketch of an implementation

In a Python implementation of this algorithm, we use the array elemer to store u_i^{n+1}, `u_1[i]` to store u_i^n, and `u_2[i]` to store u_i^{n-1}. Our convention is use `u` for the unknown new spatial field to be computed the solution at one time step back in time, `u_2` as the solution two tin back in time and so forth.

The algorithm only needs to access the three most recent time level need only three arrays for u_i^{n+1}, u_i^n, and u_i^{n-1}, $i = 0, \ldots, N_x$. Storing solutions in a two-dimensional array of size $(N_x + 1) \times (N_t + 1)$ would be

[1] `http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Dirichlet_stencil_g`
[2] `http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Dirichlet_stencil_g`

ı this simple one-dimensional PDE problem, but is normally out of the question ı three-dimensional (3D) and large two-dimensional (2D) problems. We shall ıerefore in all our programs for solving PDEs have the unknown in memory at ş few time levels as possible.

The following Python snippet realizes the steps in the computational algo- thm.

```
# Given mesh points as arrays x and t (x[i], t[n])
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx             # Courant number
Nt = len(t)-1
C2 = C**2               # Help variable in the scheme

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_1[i] - 0.5*C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

# Switch variables before next step
u_2[:], u_1[:] = u_1, u

for n in range(1, Nt):
    # Update all inner mesh points at time t[n+1]
    for i in range(1, Nx):
        u[i] = 2u_1[i] - u_2[i] - \
               C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0

    # Switch variables before next step
    u_2[:], u_1[:] = u_1, u
```

Verification

efore implementing the algorithm, it is convenient to add a source term to the DE (1) since it gives us more freedom in finding test problems for verification. ı particular, the source term allows us to use *manufactured solutions* for software esting, where we simply choose some function as solution, fit the corresponding ource term, and define boundary and initial conditions consistent with the ıosen solution. Such solutions will seldom fulfill the initial condition (3) so we eed to generalize this condition to $u_t = V(x)$.

.1 A slightly generalized model problem

Ve now address the following extended initial-boundary value problem for ne-dimensional wave phenomena:

$$
\begin{aligned}
u_{tt} &= c^2 u_{xx} + f(x,t), && x \in (0,L),\ t \in (0,T] \\
u(x,0) &= I(x), && x \in [0,L] \\
u_t(x,0) &= V(x), && x \in [0,L] \\
u(0,t) &= 0, && t > 0 \\
u(L,t) &= 0, && t > 0
\end{aligned}
$$

Sampling the PDE at (x_i, t_n) and using the same finite difference apı tions as above, yields

$$[D_t D_t u = c^2 D_x D_x + f]_i^n \,.$$

Writing this out and solving for the unknown u_i^{n+1} results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n \,.$$

The equation for the first time step must be rederived. The discretiz the initial condition $u_t = V(x)$ at $t = 0$ becomes

$$[D_{2t} u = V]_i^0 \quad \Rightarrow \quad u_i^{-1} = u_i^1 - 2\Delta t V_i,$$

which, when inserted in (23) for $n = 0$, gives the special formula

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2} C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{1}{2}\Delta t^2 f_i^n \,.$$

2.2 Using an analytical solution of physical significa

Many wave problems feature sinusoidal oscillations in time and spa example, the original PDE problem (1)-(5) allows a solution

$$u_{\mathrm{e}}(x,y,t)) = A \sin\left(\frac{\pi}{L}x\right) \cos\left(\frac{\pi}{L}ct\right) \,.$$

This u_{e} fulfills the PDE with $f = 0$, boundary conditions $u_{\mathrm{e}}(0,t) = u_{\mathrm{e}}(L$ as well as initial conditions $I(x) = A \sin\left(\frac{\pi}{L}x\right)$ and $V = 0$.

It is common to use such exact solutions of physical interest to verif mentations. However, the numerical solution u_i^n will only be an approx to $u_{\mathrm{e}}(x_i, t_n)$. We no have knowledge of the precise size of the error in this imation, and therefore we can never know if discrepancies between the co u_i^n and $u_{\mathrm{e}}(x_i, t_n)$ are caused by mathematical approximations or progr errors. In particular, if a plot of the computed solution u_i^n and the ex (25) looks similar, many are attempted to claim that the implementatio but there can still be serious programming errors although color plots lo

The only way to use exact physical solutions like (25) for serious and t verification is to run a series of finer and finer meshes, measure the int error in each mesh, and from this information estimate the convergence these rates are very close to 2, we have strong evidence that the implem works.

.3 Manufactured solution

)ne problem with the exact solution (25) is that it requires a simplification $' = 0, f = 0$) of the implemented problem (17)-(21). An advantage of using manufactured solution is that we can test all terms in the PDE problem. he idea of this approach is to set up some chosen solution and fit the source erm, boundary conditions, and initial conditions to be compatible with the iosen solution. Given that our boundary conditions in the implementation are $(0,t) = u(L,t) = 0$, we must choose a solution that fulfills these conditions.)ne example is

$$u_{\mathrm{e}}(x,t) = x(L-x)\sin t\,.$$

iserted in the PDE $u_{tt} = c^2 u_{xx} + f$ we get

$$-x(L-x)\sin t = -2\sin t + f \quad \Rightarrow f = (2 - x(L-x))\sin t\,.$$

he initial conditions become

$$\begin{aligned} u(x,0) &= I(x) = 0,\\ u_t(x,0) &= V(x) = -x(L-x)\,. \end{aligned}$$

To verify the code, we run a series of refined meshes and compute the onvergence rates. Such tests rely on an assumption that some measure E of ie numerical error is related to the discretization parameters through

$$E = C_t\Delta t^r + C_x\Delta x^p,$$

here C_t, C_x, r, and p are constants. The constants r and p are known as the *onvergence rates* in time and space, respectively. From the accuracy in the finite ifference approximations, we expect $r = p = 2$. This is confirmed by truncation ror analysis and other types of analysis. By using an exact solution of the DE problem, we can empirically compute the error measure E on a sequence f refined meshes and see if the rates $r = p = 2$ are obtained. We will not be oncerned with estimating the constants C_t and C_x.

It is advantageous to introduce a single discretization parameter $h = \Delta t = \Delta x$ for some constant $\hat{c}$ (the idea is to keep $\Delta t^r/\Delta x^p$ constant). Since Δt and x are related through the Courant number, $\Delta t = C\Delta x/c$, we set $h = \Delta t$, and ien $\Delta x = hc/C$. Now the expression for the error measure is greatly simplified:

$$E = C_t\Delta t^r + C_x\Delta x^r = C_t h^r + \frac{C_x c}{C}h^r = \hat{C}h^r, \quad \hat{C} = C_t + \frac{C_x c}{C}\,.$$

We choose an initial discretization parameter h_0 and run experiments with ecreasing h: $h_i = 2^{-i}h_0$, $i = 1, 2, \ldots, m$. Halving h in each experiment is not ecessary, but a common choice. For each experiment we must record E and

h. A standard choice of error measure is the ℓ^2 or ℓ^∞ norm of the err function e_i^n:

$$\begin{aligned} E &= ||e_i^n||_{\ell^2} = \left(\Delta t\Delta x\sum_{n=0}^{N_t}\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i,t_n) - u_i^n,\\ E &= ||e_i^n||_{\ell^\infty} = \max_{i,n}|e_n^i|\,. \end{aligned}$$

In Python, one can compute $\sum_i (e_i^{n+1})^2$ at each time step and accumu value in some sum variable, say `e2_sum`. At the final time step one `sqrt(dt*dx*e2_sum)`. For the ℓ^∞ norm one must compare the maximu at a time level (`e.max()`) with the global maximum over the time `e_max = max(e_max, e.max())`.

An alternative error measure is to use a spatial norm at one time st e.g., the end time T:

$$\begin{aligned} E &= ||e_i^n||_{\ell^2} = \left(\Delta x\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i,t_n) - u_i^n,\\ E &= ||e_i^n||_{\ell^\infty} = \max_{0\le i\le N_x}|e_n^i|\,. \end{aligned}$$

Let E_i be the error measure in experiment (mesh) number i and l the corresponding discretization parameter (h). With the error model E_i we can estimate r by comparing two consecutive experiments: $E_{i+1} =$ and $E_i = \hat{C}h_i^r$. Dividing the two equations eliminates $\hat{C}$ and solving for

$$r_i = \frac{\ln E_{i+1}/E_i}{\ln h_{i+1}/h_i}, \quad i = 0, \ldots, m-1\,.$$

We should for the present discretization method observe that r_i approac i increases.

2.4 Constructing an exact solution of the discrete tions

With a manufactured or known analytical solution, as outlined above, estimate convergence rates and see if they have the correct asymptotic b Experience shows that this is a quite good verification technique in tha common bugs will destroy the convergence rates. A significantly bet would be to check that the numerical solution is exactly what it sh This will in general require knowledge of the numerical error, which w have. However, it is possible to look for solutions where we can show t numerical error vanishes, i.e., the solution of the PDE problem is also a of the discrete equations. This property often arises if the exact solut

ower-order polynomial. (Truncation error analysis leads to error measures that nvolve derivatives of the exact solution. In the present problem, the truncation rror involves 4th-order derivatives of u in space and time. Choosing u as a olynomial of degree three or less will therefore lead to vanishing error.)

We shall now illustrate the construction of an exact solution of the PDE roblem and the discrete equations. Our choice of manufactured solution is uadratic in space and linear in time. More specifically, we set

$$u_{\mathrm{e}}(x,t) = x(L-x)(1+\frac{1}{2}t), \tag{30}$$

hich by insertion in the PDE leads to $f(x,t) = 2(1+t)c^2$. This u_{e} fulfills the oundary conditions $u = 0$ and demands $I(x) = x(L-x)$ and $V(x) = \frac{1}{2}x(L-x)$.

To realize that the chosen u_{e} is that it is also an exact solution of the discrete quations, we first establish the results

$$[D_tD_t t^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - n^2 + (n-1)^2 = 2, \tag{31}$$

$$[D_tD_t t]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - n + (n-1))\Delta t}{\Delta t^2} = 0\,. \tag{32}$$

ence,

$$[D_tD_t u_{\mathrm{e}}]_i^n = x_i(L-x_i)[D_tD_t(1+\frac{1}{2}t)]^n = x_i(L-x_i)\frac{1}{2}[D_tD_t t]^n = 0,$$

nd

$$\begin{aligned}[D_xD_x u_{\mathrm{e}}]_i^n &= (1+\frac{1}{2}t_n)[D_xD_x(xL-x^2)]_i = (1+\frac{1}{2}t_n)[LD_xD_x x - D_xD_x x^2]_i \\ &= -2(1+\frac{1}{2}t_n)\,.\end{aligned}$$

ow, $f_i^n = 2(1+\frac{1}{2}t_n)c^2$ and we get

$$[D_tD_t u_{\mathrm{e}} - c^2 D_xD_x u_{\mathrm{e}} - f]_i^n = 0 - c^2(-1)2(1+\frac{1}{2}t_n + 2(1+\frac{1}{2}t_n)c^2 = 0\,.$$

Moreover, $u_{\mathrm{e}}(x_i,0) = I(x_i)$, $\partial u_{\mathrm{e}}/\partial t = V(x_i)$ at $t = 0$, and $u_{\mathrm{e}}(x_0,t) = {}_{\mathrm{e}}(x_{N_x},0) = 0$. Also the modified scheme for the first time step is fulfilled by ${}_{\mathrm{e}}(x_i,t_n)$.

Therefore, the exact solution $u_{\mathrm{e}}(x,t) = x(L-x)(1+t/2)$ of the PDE problem also an exact solution of the discrete problem. We can use this result to check hat the computed u_i^n vales from an implementation equals $u_{\mathrm{e}}(x_i,t_n)$ within achine precision, *regardless of the mesh spacings* Δx and Δt! Nevertheless, here might be stability restrictions on Δx and Δt, so the test can only be run or a mesh that is compatible with the stability criterion (which in the present ase is $C \leq 1$, to be derived later).

Notice.

A product of quadratic or linear expressions in the various indepen variables, as shown above, will often fulfill both the continuous and dis PDE problem and can therefore be very useful solutions for veri implementations. However, for 1D wave equations of the type $u_t =$ we shall see that there is always another much more powerful w generating exact solutions (just set $C = 1$).

3 Implementation

This section present the complete computational algorithm, its implement Python code, animation of the solution, and verification of the impleme

A real implementation of the basic computational algorithm from Sect and 1.6 can be encapsulated in a function, taking all the input data problem as arguments. The physical input data consists of c, $I(x)$, $V(x)$ L, and T. The numerical input is the mesh parameters Δt and Δx.

Instead of specifying Δt *and* Δx, we can specify one of them and the number C instead, since having explicit control of the Courant nu convenient when investigating the numerical method. Many find it na prescribe the resolution of the spatial grid and set N_x. The solver f can then compute $\Delta t = CL/(cN_x)$. However, for comparing $u(x,t)$ cu functions of x) for various Courant numbers, especially in animations it is more convenient to keep Δt fixed for all C and let Δx vary acco $\Delta x = c\Delta t/C$. (With Δt fixed, all frames correspond to the same tim plotting curves with different spatial resolution is trivial.)

The solution at all spatial points at a new time level is stored in a `u` (of length N_x+1). We need to decide what do to with this soluti visualize the curve, analyze the values, or write the array to file for l The decision what to do is left to the user in a suppled function

```
def user_action(u, x, t, n):
```

where `u` is the solution at the spatial points `x` at time `t[n]`.

3.1 Making a solver function

A first attempt at a solver function is listed below.

```
from numpy import *

def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = linspace(0, Nt*dt, Nt+1)   # Mesh points in time
    dx = dt*c/float(C)
```

```
    Nx = int(round(L/dx))
    x = linspace(0, L, Nx+1)       # Mesh points in space
    C2 = C**2                      # Help variable in the scheme
    if f is None or f == 0 :
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    u   = zeros(Nx+1)   # Solution array at new time level
    u_1 = zeros(Nx+1)   # Solution at 1 time level back
    u_2 = zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # for measuring CPU time

    # Load initial condition into u_1
    for i in range(0,Nx+1):
        u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, 0)

    # Special formula for first time step
    n = 0
    for i in range(1, Nx):
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               0.5*dt**2*f(x[i], t[n])
    u[0] = 0;  u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    # Switch variables before next step
    u_2[:] = u_1;  u_1[:] = u

    for n in range(1, Nt):
        # Update all inner points at time t[n+1]
        for i in range(1, Nx):
            u[i] = - u_2[i] + 2*u_1[i] + \
                     C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                     dt**2*f(x[i], t[n])

        # Insert boundary conditions
        u[0] = 0;  u[Nx] = 0
        if user_action is not None:
            if user_action(u, x, t, n+1):
                break

        # Switch variables before next step
        u_2[:] = u_1;  u_1[:] = u

    cpu_time = t0 - time.clock()
    return u, x, t, cpu_time
```

.2 Verification: exact quadratic solution

We use the test problem derived in Section 2.1 for verification. Here is a function realizing this verification as a nose test:


```
import nose.tools as nt

def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""
    def u_exact(x, t):
        return x*(L-x)*(1 + 0.5*t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5*u_exact(x, 0)

    def f(x, t):
        return 2*(1 + 0.5*t)*c**2

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T)
    u_e = u_exact(x, t[-1])
    diff = abs(u - u_e).max()
    nt.assert_almost_equal(diff, 0, places=14)
```

3.3 Visualization: animating the solution

Now that we have verified the implementation it is time to do a real comp where we also display the evolution of the waves on the screen.

Visualization via SciTools. The following `viz` function defines a user callback function for plotting the solution at each time level:

```
def viz(I, V, f, c, L, dt, C, T, umin, umax, animate=True):
    """Run solver and visualize u at each time level."""
    import scitools.std as plt
    import time, glob, os

    def plot_u(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, L, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Let the initial condition stay on the screen for 2
        # seconds, else insert a pause of 0.2 s between each plot
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n)  # for movie making

    # Clean up old movie frames
    for filename in glob.glob('frame_*.png'):
        os.remove(filename)
```

```
    user_action = plot_u if animate else None
    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T, user_action)

    # Make movie files
    fps = 4  # Frames per second
    plt.movie('frame_*.png', encoder='html', fps=fps,
              output_file='movie.html')
    codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                     libtheora='ogg')
    filespec = 'frame_%04d.png'
    movie_program = 'avconv'  # or 'ffmpeg'
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
              '-vcodec %(codec)s movie.%(ext)s' % vars()
        os.system(cmd)
```

function inside another function, like `plot_u` in the above code segment, has ccess to *and remembers* all the local variables in the surrounding code inside ıe `viz` function (!). This is known in computer science as a *closure* and is ery convenient to program with. For example, the `plt` and `time` modules efined outside `plot_u` are accessible for `plot_u` when the function is called (as `ser_action`) in the `solver` function. Some may think, however, that a class ıstead of a closure is a cleaner and easier-to-understand implementation of the ser action function, see Section 8.

Making movie files. Several hardcopies of the animation are made from ıe `frame_*.png` files. We use the `avconv` (or `ffmpeg`) programs to combine ıdividual plot files to movies in modern formats: Flash, MP4, Webm, and)gg. A typical `avconv` (or `ffmpeg`) command for creating a movie file in Ogg ›rmat with 4 frames per second built from a collection of plot files with names enerated by `frame_%04d.png`, look like

```
erminal> avconv -r 4 -i frame_%04d.png -c:v libtheora movie.ogg
```

he different formats require different video encoders (`-c:v`) to be installed: lash applies `flv`, WebM applies `libvpx`, and MP4 applies `libx264`:

```
erminal> avconv -r 4 -i frame_%04d.png -c:v flv movie.flv
erminal> avconv -r 4 -i frame_%04d.png -c:v libvpx movie.webm
erminal> avconv -r 4 -i frame_%04d.png -c:v libx264 movie.mp4
```

Players like `vlc`, `mplayer`, `gxine`, and `totem` can be used to play these movie les.

Note that padding the frame counter with zeros in the `frame_*.png` files, s specified by the `%04d` format, is essential so that the wildcard notation `rame_*.png` expands to the correct set of files.

The `plt.movie` function also creates a `movie.html` file with a movi for displaying the `frame_*.png` files in a web browser. This movie playe generated from the command line too

```
Terminal> scitools movie encoder=html output_file=movie.html \
          fps=4 frame_*.png
```

Skipping frames for animation speed. Sometimes the time step and T is large, leading to an inconveniently large number of plot files an animation on the screen. The solution to such a problem is to decide or number of frames in the animation, `num_frames`, and plot the solution every `every` frame. The total number of time levels (i.e., maximum number of frames) is the length of `t`, `t.size`, and if we want `num_fra` need to plot every `t.size/num_frames` frame:

```
every = int(t.size/float(num_frames))
if n % every == 0 or n == t.size-1:
    st.plot(x, u, 'r-', ...)
```

The initial condition (`n=0`) is natural to include, and as `n % every =` very seldom be true for the very final frame, we also ensure that `n == t.` and hence the final frame is included.

A simple choice of numbers may illustrate the formulas: say we h frames in total (`t.size`) and we allow only 60 frames to be plotted. T need to plot every 801/60 frame, which with integer division yields 13 a Using the mod function, `n % every`, this operation is zero every time `n` divided by 13 without a remainder. That is, the `if` test is true when `n` $0, 13, 26, 39, ..., 780, 801$. The associated code is included in the `plot_u` in the file `wave1D_u0v.py`[3].

Visualization via Matplotlib. The previous code based on the `plot` i from `scitools.std` can be run with Matplotlib as the visualization b but if one desires to program directly with Matplotlib, quite different needed. Matplotlib's interactive mode must be turned on:

```
import matplotlib.pyplot as plt
plt.ion()  # interactive mode on
```

The most commonly used animation technique with Matplotlib is to up data in the plot at each time level:

```
# Make a first plot
lines = plt.plot(t, u)
# call plt.axis, plt.xlabel, plt.ylabel, etc. as desired
```

[3] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0v.py`

```
# At later time levels
lines[0].set_ydata(u)
plt.legend('t=%g' % t[n])
plt.draw()  # make updated plot
plt.savefig(...)
```

An alternative is to rebuild the plot at every time level:

```
plt.clf()        # delete any previous curve(s)
plt.axis([...])
plt.plot(t, u)
# plt.xlabel, plt.legend and other decorations
plt.draw()
plt.savefig(...)
```

Many prefer to work with figure and axis objects as in MATLAB:

```
fig = plt.figure()
...
fig.clf()
ax = fig.gca()
ax.axis(...)
ax.plot(t, u)
# ax.set_xlabel, ax.legend and other decorations
plt.draw()
fig.savefig(...)
```

.4 Running a case

The first demo of our 1D wave equation solver concerns vibrations of a string that is initially deformed to a triangular shape, like when picking a guitar string:

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{33}$$

We choose $L = 75$ cm, $x_0 = 0.8L$, $a = 5$ mm, and a time frequency $\nu = 440$ Hz. The relation between the wave speed c and ν is $c = \nu\lambda$, where λ is the wavelength, taken as $2L$ because the longest wave on the string form half a wavelength. There is no external force, so $f = 0$, and the string is at rest initially so that $V = 0$.

Regarding numerical parameters, we need to specify a Δt. Sometimes it is more natural to think of a spatial resolution instead of a time step. A natural semi-coarse spatial resolution in the present problem is $N_x = 50$. We can then choose the associated Δt (as required by the `viz` and `solver` functions) as the stability limit: $\Delta t = L/(N_x c)$. This is the Δt to be specified, but notice that if $C < 1$, the actual Δx computed in `solver` gets larger than L/N_x: $\Delta x = c\Delta t/C = L/(N_x C)$. (The reason is that we fix Δt and adjust Δx, so if C gets smaller, the code implements this effect in terms of a larger Δx.)

A function for setting the physical and numerical parameters and calling `viz` in this application goes as follows:


```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

    def I(x):
        return a*x/x0 if x < x0 else a/(L-x0)*(L-x)

    umin = -1.2*a;  umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)
```

The associated program has the name `wave1D_u0.py`[4]. Run the progr watch the movie of the vibrating string[5].

3.5 The benefits of scaling

The previous example demonstrated that quite some work is needed tablishing relevant physical parameters for a case. By *scaling* the mathe problem we can often reduce the need to estimate physical parameters cally. A scaling consists of introducing new independent and dependent v with the aim that the absolute value of these vary between 0 and 1:

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{c}{L}t, \quad \bar{u} = \frac{u}{a}.$$

Replacing old by new variables in the PDE, using $f = 0$, and dropping t results in the *scaled equation* $u_{tt} = u_{xx}$. This equation has no physical pa (!).

If we have a program implemented for the physical wave equati dimensions, we can obtain the dimensionless, scaled version by settin The initial condition corresponds to (185), but with setting $a = 1$, $L =$ $x_0 \in [0, 1]$. This means that we only need to decide on the x_0 value as a of unity, because the scaled problem corresponds to setting all other par to unity! In the code we can just set `a=c=L=1`, `x0=0.8`, and there is no calculate with wavelengths and frequencies to estimate c.

The only non-trivial parameter to estimate in the scaled problem is t end time of the simulation, or more precisely, how it relates to periods in solutions in time, since we often want to express the end time as a number of periods. Suppose as u behaves as $\sin(\omega t)$ in time in variab dimension. The corresponding period is $P = 2\pi/\omega$. The frequency ω is re

[4] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0.py`
[5] `http://tinyurl.com/opdfafk/pub/mov-wave/guitar_C0.8/index.html`

ıe wavelength λ of the waves through the relations $\omega = kc$ and $k = 2\pi/\lambda$, giving
$= 2\pi c/\lambda$ and $P = \lambda/c$. It remains to estimate λ. With $u(x,t) = F(x)\sin\omega t$
e find from $u_{tt} = c^2 u_{xx}$ that $c^2F'' + \omega^2 F = 0$, and the boundary conditions
emand $F(0) = F(L) = 0$. The solution is $F(x) = \sin(x\pi/L)$, which has
avelength $\lambda = 2\pi/(\pi/L) = 2L$. One period is therefore given by $P = 2L/c$.
he dimensionless period is $\bar{P} = Pc/L = 2$.

: Vectorization

he computational algorithm for solving the wave equation visits one mesh
oint at a time and evaluates a formula for the new value u_i^{n+1} at that point.
echnically, this is implemented by a loop over array elements in a program.
uch loops may run slowly in Python (and similar interpreted languages such as
, and MATLAB). One technique for speeding up loops is to perform operations
n entire arrays instead of working with one element at a time. This is referred
› as *vectorization*, *vector computing*, or *array computing*. Operations on whole
rrays are possible if the computations involving each element is independent of
ach other and therefore can, at least in principle, be performed simultaneously.
ectorization not only speeds up the code on serial computers, but it also makes
easy to exploit parallel computing.

.1 Operations on slices of arrays

fficient computing with `numpy` arrays demands that we avoid loops and compute
ith entire arrays at once (or at least large portions of them). Consider this
alculation of differences $d_i = u_{i+1} - u_i$:

```
ı = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

ll the differences here are independent of each other. The computation of `d` can
ıerefore alternatively be done by subtracting the array $(u_0, u_1, \ldots, u_{n-1})$ from
ıe array where the elements are shifted one index upwards: $(u_1, u_2, \ldots, u_n)$,
ee Figure 3. The former subset of the array can be expressed by `u[0:n-1]`,
`[0:-1]`, or just `u[:-1]`, meaning from index 0 up to, but not including, the
ıst element (`-1`). The latter subset is obtained by `u[1:n]` or `u[1:]`, meaning
om index 1 and the rest of the array. The computation of `d` can now be done
ithout an explicit Python loop:

```
ł = u[1:] - u[:-1]
```

r with explicit limits if desired:

```
ł = u[1:n] - u[0:n-1]
```


Indices with a colon, going from an index to (but not including) anothe
are called *slices*. With `numpy` arrays, the computations are still done b
but in efficient, compiled, highly optimized code in C or Fortran. Suc
operations can also easily be distributed among many processors on
computers. We say that the *scalar code* above, working on an element (a
at a time, has been replaced by an equivalent *vectorized code*. The pr
vectorizing code is called *vectorization*.

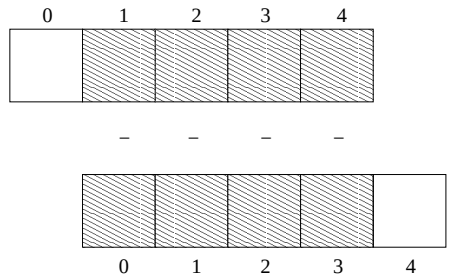


Figure 3: Illustration of subtracting two slices of two arrays.

Test the understanding.

Newcomers to vectorization are encouraged to choose a small array u
with five elements, and simulate with pen and paper both the loop ve
and the vectorized version.

Finite difference schemes basically contains differences between array e
with shifted indices. Consider the updating formula

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

The vectorization consists of replacing the loop by arithmetics on slices c
of length `n-2`:

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

Note that `u2` here gets length `n-2`. If `u2` is already an array of lengt
we want to use the formula to update all the "inner" elements of `u2`, as
when solving a 1D wave equation, we can write

```
u2[1:-1]  = u[:-2] - 2*u[1:-1] + u[2:]
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

Pen and paper calculations with a small array will demonstrate what is
going on. The expression on the right-hand side are done in the followir

nvolving temporary arrays with intermediate results, since we can only work ith two arrays at a time in arithmetic expressions:

```
emp1 = 2*u[1:-1]
emp2 = u[0:-2] - temp1
emp3 = temp2 + u[2:]
u2[1:-1] = temp3
```

We can extend the previous example to a formula with an additional term omputed by calling a function:

```
def f(x):
    return x**2 + 1

for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])
```

ssuming `u2`, `u`, and `x` all have length `n`, the vectorized version becomes

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:] + f(x[1:-1])
```

.2 Finite difference schemes expressed as slices

We now have the necessary tools to vectorize the algorithm for the wave equation. here are three loops: one for the initial condition, one for the first time step, nd finally the loop that is repeated for all subsequent time levels. Since only he latter is repeated a potentially large number of times, we limit the efforts of ectorizing the code to this loop:

```
for i in range(1, Nx):
    u[i] = 2*u_1[i] - u_2[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

he vectorized version becomes

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

```
u[1:Nx] = 2*u_1[1:Nx]- u_2[1:Nx] + \
          C2*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1])
```

The program `wave1D_u0v.py`[6] contains a new version of the function `solver` here both the scalar and the vectorized loops are included (the argument `ersion` is set to `scalar` or `vectorized`, respectively).

[6] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0v.py`

4.3 Verification

We may reuse the quadratic solution $u_e(x,t) = x(L-x)(1+\frac{1}{2}t)$ for verify the vectorized code. A nose test can now test both the scalar and the ve version. Moreover, we may use a `user_action` function that compa computed and exact solution at each time level and performs a test:

```
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    u_exact = lambda x, t: x*(L - x)*(1 + 0.5*t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5*u_exact(x, 0)
    # f is a scalar (zeros_like(x) works for scalar x too)
    f = lambda x, t: zeros_like(x) + 2*c**2*(1 + 0.5*t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = abs(u - u_e).max()
        nt.assert_almost_equal(diff, 0, places=13)

    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='scalar')
    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='vectorized')
```

Lambda functions.

The code segment above demonstrates how to achieve very compact with the use of lambda functions for the various input parameters require a Python function. In essence,

```
f = lambda x, t: L*(x-t)**2
```

is equivalent to

```
def f(x, t):
    return L(x-t)**2
```

Note that lambda functions can just contain a single expression an statements.

One advantage with lambda functions is that they can be used directly in calls:

```
solver(I=lambda x: sin(pi*x/L), V=0, f=0, ...)
```

.4 Efficiency measurements

unning the `wave1D_u0v.py` code with the previous string vibration exam-le for $N_x = 50, 100, 200, 400, 800$, and measuring the CPU time (see the `un_efficiency_experiments` function), shows that the vectorized code runs ubstantially faster: the scalar code uses approximately a factor $N_x/5$ more me!

.5 Remark on the updating of arrays

t the end of each time step we need to update the `u_2` and `u_1` arrays such hat they have the right content for the next time step:

```
_2[:] = u_1
_1[:] = u
```

he order here is important! (Updating `u_1` first, makes `u_2` equal to `u`, which wrong.)

The assignment `u_1[:] = u` copies the content of the `u` array into the ements of the `u_1` array. Such copying takes time, but little compared to omputing `u` from the finite difference formula, even when the formula has a ectorized implementation. However, efficiency of program code is a key topic hen solving PDEs numerically, so it must be mentioned that there exists a uch more efficient way of making the arrays `u_2` and `u_1` ready for the next me step. The idea is based on *switching references* and explained below.

A Python variable is actually a reference to some object (C programmers ay think of pointers). Instead of copying data, we can let `u_2` refer to the `u_1` bject and `u_1` refer to the `u` object. A naive implementation like

```
_2 = u_1
_1 = u
```

ill fail, however, because now `u_1` and `u` refers to the same object and the pdate of `u` from the finite difference formula at the next time step will overwrite _1 and lead to erroneous computations. Also, with the suggested change of eferences, the reference to the `u_2` array is lost, and we have in fact only two rrays. The solution to this problem is to ensure that `u` points to the `u_2` array. his is mathematically wrong, but new correct values will be filled into `u` at the ext time step.

The correct switch of references is then

```
tmp = u_2
u_2 = u_1
u_1 = u
u = tmp
```

We can get rid of the temporary reference `tmp` by writing

```
u_2, u_1, u = u_1, u, u_2
```

This update will be used in later implementations.

Caution:

The update `u_2, u_1, u = u_1, u, u_2` leaves wrong content in `u` a final time step. This means that if we return `u`, as we do in the exa codes here, we actually return `u_2`, which is obviously wrong. It is ther important to adjust the content of `u` to `u = u_1` before returning `u`.

5 Exercises

Exercise 1: Simulate a standing wave

The purpose of this exercise is to simulate standing waves on $[0, L]$ and il the error in the simulation. Standing waves arise from an initial condit

$$u(x, 0) = A \sin\left(\frac{\pi}{L} mx\right),$$

where m is an integer and A is a freely chosen amplitude. The corresp exact solution can be computed and reads

$$u_{\mathrm{e}}(x, t) = A \sin\left(\frac{\pi}{L} mx\right) \cos\left(\frac{\pi}{L} mct\right).$$

a) Explain that for a function $\sin kx \cos \omega t$ the wave length in space is λ and the period in time is $P = 2\pi/\omega$. Use these expressions to find tl length in space and period in time of u_{e} above.

b) Import the `solver` function `wave1D_u0.py` into a new file where function is reimplemented such that it plots either the numerical *and* tl solution, *or* the error.

c) Make animations where you illustrate how the error $e_i^n = u_{\mathrm{e}}(x_i, t$ develops and increases in time. Also make animations of u and u_{e} simulta

Hint 1. Quite long time simulations are needed in order to display sig discrepancies between the numerical and exact solution.

Hint 2. A possible set of parameters is $L = 12$, $m = 9$, $c = 2$, $A = 1$, $N_x = 80$, $' = 0.8$. The error mesh function e^n can be simulated for 10 periods, while 0-30 periods are needed to show significant differences between the curves for ıe numerical and exact solution.
ilename: `wave_standing.py`.

:emarks. The important parameters for numerical quality are C and $k\Delta x$, here $C = c\Delta t/\Delta x$ is the Courant number and k is defined above ($k\Delta x$ is roportional to how many mesh points we have per wave length in space, see ection 10.4 for explanation).

:xercise 2: Add storage of solution in a user action function

xtend the `plot_u` function in the file `wave1D_u0.py` to also store the solutions in a list. To this end, declare `all_u` as an empty list in the `viz` function, utside `plot_u`, and perform an append operation inside the `plot_u` function. ote that a function, like `plot_u`, inside another function, like `viz`, remembers ll local variables in `viz` function, including `all_u`, even when `plot_u` is called ıs `user_action`) in the `solver` function. Test both `all_u.append(u)` and `ll_u.append(u.copy())`. Why does one of these constructions fail to store ıe solution correctly? Let the `viz` function return the `all_u` list converted to a wo-dimensional `numpy` array. Filename: `wave1D_u0_s_store.py`.

:xercise 3: Use a class for the user action function

edo Exercise 2 using a class for the user action function. That is, define a class `ction` where the `all_u` list is an attribute, and implement the user action func- on as a method (the special method `__call__` is a natural choice). The class ersions avoids that the user action function depends on parameters defined out- de the function (such as `all_u` in Exercise 2). Filename: `wave1D_u0_s2c.py`.

:xercise 4: Compare several Courant numbers in one movie

he goal of this exercise is to make movies where several curves, corresponding) different Courant numbers, are visualized. Import the `solver` function from ıe `wave1D_u0_s` movie in a new file `wave_compare.py`. Reimplement the `viz` unction such that it can take a list of `C` values as argument and create a movie ith solutions corresponding to the given `C` values. The `plot_u` function must e changed to store the solution in an array (see Exercise 2 or 3 for details), `olver` must be computed for each value of the Courant number, and finally ıe must run through each time step and plot all the spatial solution curves in ıe figure and store it in a file.

The challenge in such a visualization is to ensure that the curves in one plot orresponds to the same time point. The easiest remedy is to keep the time and pace resolution constant and change the wave velocity c to change the Courant umber. Filename: `wave_numerics_comparison.py`.

Project 5: Calculus with 1D mesh functions

This project explores integration and differentiation of mesh functions, bc scalar and vectorized implementations. We are given a mesh function spatial one-dimensional mesh $x_i = i\Delta x$, $i = 0, \ldots, N_x$, over the interval

a) Define the discrete derivative of f_i by using centered differences at mesh points and one-sided differences at the end points. Implement version of the computation in a Python function and supply a nose test linear case $f(x) = 4x - 2.5$ where the discrete derivative should be exa

b) Vectorize the implementation of the discrete derivative. Extend the r to check the validity of the implementation.

c) To compute the discrete integral F_i of f_i, we assume that the mesh f f_i varies linearly between the mesh points. Let $f(x)$ be such a linear int of f_i. We then have

$$F_i = \int_{x_0}^{x_i} f(x)dx\,.$$

The exact integral of a piecewise linear function $f(x)$ is given by the Tra rule. S how that if F_i is already computed, we can find F_{i+1} from

$$F_{i+1} = F_i + \frac{1}{2}(f_i + f_{i+1})\Delta x\,.$$

Make a function for a scalar implementation of the discrete integral as function. That is, the function should return F_i for $i = 0, \ldots, N_x$. Fo test one can use the fact that the above defined discrete integral of function (say $f(x) = 4x - 2.5$) is exact.

d) Vectorize the implementation of the discrete integral. Extend the n to check the validity of the implementation.

Hint. Interpret the recursive formula for F_{i+1} as a sum. Make an arr each element of the sum and use the "cumsum" (`numpy.cumsum`) oper compute the accumulative sum: `numpy.cumsum([1,3,5])` is `[1,4,9]`.

e) Create a class `MeshCalculus` that can integrate and differentiate me tions. The class can just define some methods that call the previousl mented Python functions. Here is an example on the usage:

```
import numpy as np
calc = MeshCalculus(vectorized=True)
x = np.linspace(0, 1, 11)        # mesh
f = np.exp(x)                    # mesh function
df = calc.differentiate(f, x)    # discrete derivative
F = calc.integrate(f, x)         # discrete anti-derivative
```

Filename: `mesh_calculus_1D.py`.

5 Generalization: reflecting boundaries

he boundary condition $u = 0$ makes u change sign at the boundary, while ıe condition $u_x = 0$ perfectly reflects the wave, see a web page[7] or a movie le[8] for demonstration. Our next task is to explain how to implement the oundary condition $u_x = 0$, which is more complicated to express numerically nd also to implement than a given value of u. We shall present two methods or implementing $u_x = 0$ in a finite difference scheme, one based on deriving a ıodified stencil at the boundary, and another one based on extending the mesh ith ghost cells and ghost points.

.1 Neumann boundary condition

/hen a wave hits a boundary and is to be reflected back, one applies the ɔndition

$$\frac{\partial u}{\partial n} \equiv \boldsymbol{n} \cdot \nabla u = 0\,. \tag{34}$$

he derivative $\partial/\partial n$ is in the outward normal direction from a general boundary. or a 1D domain $[0, L]$, we have that

$$\left.\frac{\partial}{\partial n}\right|_{x=L} = \frac{\partial}{\partial x}, \quad \left.\frac{\partial}{\partial n}\right|_{x=0} = -\frac{\partial}{\partial x}\,.$$

Boundary condition terminology.

Boundary conditions that specify the value of $\partial u/\partial n$, or shorter u_n, are known as Neumann[a] conditions, while Dirichlet conditions[b] refer to specifications of u. When the values are zero ($\partial u/\partial n = 0$ or $u = 0$) we speak about *homogeneous* Neumann or Dirichlet conditions.

[a] http://en.wikipedia.org/wiki/Neumann_boundary_condition
[b] http://en.wikipedia.org/wiki/Dirichlet_conditions

.2 Discretization of derivatives at the boundary

ow can we incorporate the condition (34) in the finite difference scheme? Since e have used central differences in all the other approximations to derivatives ı the scheme, it is tempting to implement (34) at $x = 0$ and $t = t_n$ by the ifference

$$\frac{u^n_{-1} - u^n_1}{2\Delta x} = 0\,. \tag{35}$$

[7] http://tinyurl.com/opdfafk/pub/mov-wave/demo_BC_gaussian/index.html
[8] http://tinyurl.com/opdfafk/pub/mov-wave/demo_BC_gaussian/movie.flv

The problem is that u^n_{-1} is not a u value that is being computed since tl is outside the mesh. However, if we combine (35) with the scheme for i

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + C^2\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right),$$

we can eliminate the fictitious value u^n_{-1}. We see that $u^n_{-1} = u^n_1$ from (35 can be used in (36) to arrive at a modified scheme for the boundary poi

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + 2C^2\left(u^n_{i+1} - u^n_i\right), \quad i = 0\,.$$

Figure 4 visualizes this equation for computing u^3_0 in terms of u^2_0, u^1_0, a

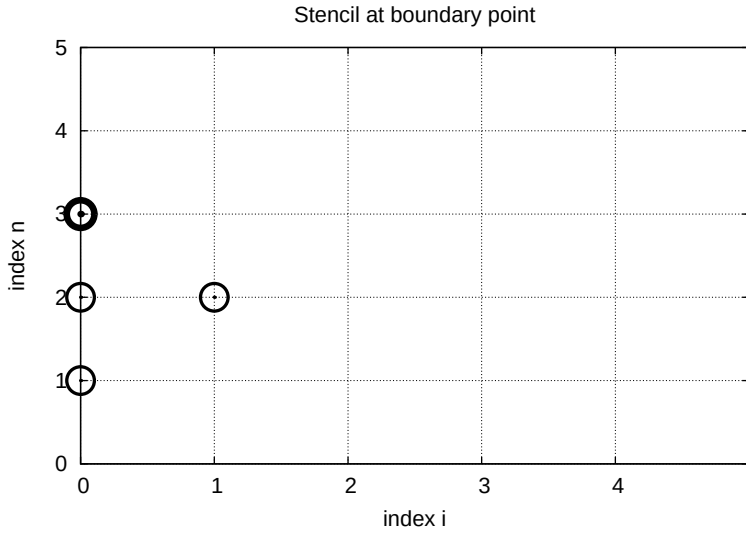


Figure 4: Modified stencil at a boundary with a Neumann conditi

Similarly, (34) applied at $x = L$ is discretized by a central differenc

$$\frac{u^n_{N_x+1} - u^n_{N_x-1}}{2\Delta x} = 0\,.$$

Combined with the scheme for $i = N_x$ we get a modified scheme for the b value $u^{n+1}_{N_x}$:

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + 2C^2\left(u^n_{i-1} - u^n_i\right), \quad i = N_x\,.$$

The modification of the scheme at the boundary is also required special formula for the first time step. How the stencil moves through tl and is modified at the boundary can be illustrated by an animation i page[9] or a movie file[10].

[9] http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Neumann_stencil_gpl
[10] http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Neumann_stencil_gpl

.3 Implementation of Neumann conditions

he implementation of the special formulas for the boundary points can benefit om using the general formula for the interior points also at the boundaries, ut replacing u^n_{i-1} by u^n_{i+1} when computing u^{n+1}_i for $i = 0$ and u^n_{i+1} by u^n_{i-1} for $= N_x$. This is achieved by just replacing the index $i-1$ by $i+1$ for $i = 0$ and $+1$ by $i-1$ for $i = N_x$. In a program, we introduce variables to hold the value f the offset indices: `im1` for `i-1` and `ip1` for `i+1`. It is now just a manner of efining `im1` and `ip1` properly for the internal points and the boundary points. he coding for the latter reads

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

We can in fact create one loop over both the internal and boundary points nd use only one updating formula:

```
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

The program `wave1D_n0.py`[11] contains a complete implementation of the D wave equation with boundary conditions $u_x = 0$ at $x = 0$ and $x = L$.

It would be nice to modify the `test_quadratic` test case from the `wave1D_u0.py` ith Dirichlet conditions, described in Section 4.3. However, the Neumann conitions requires the polynomial variation in x directory to be of third degree, hich causes challenging problems with designing a test where the numerical olution is known exactly. Exercise 10 outlines ideas and code for this purpose. he only test in `wave1D_n0.py` is to start with a plug wave at rest and see that ne initial condition is reached again perfectly after one period of motion, if $' = 1$.

.4 Index set notation

We shall introduce a special notation for index sets, consisting of writing x_i, $\in \mathcal{I}_x$, instead of $i = 0, \ldots, N_x$. Obviously, $\mathcal{I}_x$ must be the set $\mathcal{I}_x = \{0, \ldots, N_x\}$, ut it is often advantageous to have a symbol for this set rather than specifying ll its elements. This saves writing and makes specification of algorithms and nplementation of computer code easier.

[11] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0.py`

The first index in the set will be denoted $\mathcal{I}^0_x$ and the last $\mathcal{I}^{-1}_x$. Somet need to count from the second element in the set, and the notation $\mathcal{I}^+_x$ used. Correspondingly, $\mathcal{I}^-_x$ means $\{0, \ldots, N_x - 1\}$. All the indices corres to inner grid points are $\mathcal{I}^i_x = \{1, \ldots, N_x - 1\}$. For the time domain w natural to explicitly use 0 as the first index, so we will usually write n t_0 rather than $n = \mathcal{I}^0_t$. We also avoid notation like $x_{\mathcal{I}^{-1}_x}$ and will instead $i = \mathcal{I}^{-1}_x$.

The Python code associated with index sets applies the following conv

Notation	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}^0_x$	`Ix[0]`
$\mathcal{I}^{-1}_x$	`Ix[-1]`
$\mathcal{I}^-_x$	`Ix[:-1]`
$\mathcal{I}^+_x$	`Ix[1:]`
$\mathcal{I}^i_x$	`Ix[1:-1]`

An important feature of the index set notation is that it keeps our formu code independent of how we count mesh points. For example, the notatio or $i = \mathcal{I}^0_x$ remains the same whether $\mathcal{I}_x$ is defined as above or as starti i.e., $\mathcal{I}_x = \{1, \ldots, Q\}$. Similarly, we can in the code define `Ix=range(N` `Ix=range(1,Q)`, and expressions like `Ix[0]` and `Ix[1:-1]` remain corre application where the index set notation is convenient is conversion of co a language where arrays has base index 0 (e.g., Python and C) to la where the base index is 1 (e.g., MATLAB and Fortran). Another im application is implementation of Neumann conditions via ghost points (section).

For the current problem setting in the x, t plane, we work with the in

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\},$$

defined in Python as

```
Ix = range(0, Nx+1)
It = range(0, Nt+1)
```

A finite difference scheme can with the index set notation be specifi

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + C^2\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right), \quad i \in \mathcal{I}^i_x,\ n \in \mathcal{I}_t$$
$$u_i = 0, \quad i = \mathcal{I}^0_x,\ n \in \mathcal{I}^i_t,$$
$$u_i = 0, \quad i = \mathcal{I}^{-1}_x,\ n \in \mathcal{I}^i_t,$$

and implemented by code like

```
for n in It[1:-1]:
    for i in Ix[1:-1]:
        u[i] = - u_2[i] + 2*u_1[i] + \
               C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Notice.

The program `wave1D_dn.py`[a] applies the index set notation and solves the 1D wave equation $u_{tt} = c^2 u_{xx} + f(x,t)$ with quite general boundary and initial conditions:

- $x = 0$: $u = U_0(t)$ or $u_x = 0$
- $x = L$: $u = U_L(t)$ or $u_x = 0$
- $t = 0$: $u = I(x)$
- $t = 0$: $u_t = I(x)$

The program combines Dirichlet and Neumann conditions, scalar and vectorized implementation of schemes, and the index notation into one piece of code. A lot of test examples are also included in the program:

- A rectangular plug profile as initial condition (easy to use as test example as the rectangle should jump one cell per time step when $C = 1$, without any numerical errors).
- A Gaussian function as initial condition.
- A triangular profile as initial condition, which resembles the typical initial shape of a guitar string.
- A sinusoidal variation of u at $x = 0$ and either $u = 0$ or $u_x = 0$ at $x = L$.
- An exact analytical solution $u(x,t) = \cos(m\pi t/L)\sin(\frac{1}{2}m\pi x/L)$, which can be used for convergence rate tests.

[a] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn.py`

.5 Alternative implementation via ghost cells

Idea. Instead of modifying the scheme at the boundary, we can introduce extra points outside the domain such that the fictitious values u^n_{-1} and $u^n_{N_x+1}$ are

defined in the mesh. Adding the intervals $[-\Delta x, 0]$ and $[L, L+\Delta x]$, often to as *ghost cells*, to the mesh gives us all the needed mesh points, corres to $i = -1, 0, \ldots, N_x, N_x+1$. The extra points $i = -1$ and $i = N_x+1$ ar as *ghost points*, and values at these points, u^n_{-1} and $u^n_{N_x+1}$, are calle *values*.

The important idea is to ensure that we always have

$$u^n_{-1} = u^n_1 \text{ and } u^n_{N_x+1} = u^n_{N_x-1},$$

because then the application of the standard scheme at a boundary poi or $i = N_x$ will be correct and guarantee that the solution is compatible boundary condition $u_x = 0$.

Implementation. The `u` array now needs extra elements correspon the ghost cells and points. Two new point values are needed:

```
u   = zeros(Nx+3)
```

The arrays `u_1` and `u_2` must be defined accordingly.

Unfortunately, a major indexing problem arises with ghost cells. Th is that Python indices *must* start at 0 and `u[-1]` will always mean element in `u`. This fact gives, apparently, a mismatch between the mathe indices $i = -1, 0, \ldots, N_x+1$ and the Python indices running over `u`: `0,.` One remedy is to change the mathematical notation of the scheme, as i

$$u^{n+1}_i = \cdots, \quad i = 1, \ldots, N_x+1,$$

meaning that the ghost points correspond to $i = 0$ and $i = N_x+1$. A solution is to use the ideas of Section 6.4: we hide the specific index an index set and operate with inner and boundary points using the i notation.

To this end, we define `u` with proper length and `Ix` to be the corres indices for the real physical points $(1, 2, \ldots, N_x+1)$:

```
u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)
```

That is, the boundary points have indices `Ix[0]` and `Ix[-1]` (as befo first update the solution at all physical mesh points (i.e., interior point mesh extended with ghost cells):

```
for i in Ix:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

It remains to update the ghost points. For a boundary condition u_x = ghost value must equal to the value at the associated inner mesh point. C code makes this statement precise:

```
i = Ix[0]           # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1]          # x=L boundary
u[i+1] = u[i-1]
```

The physical solution to be plotted is now in `u[1:-1]`, or equivalently `[Ix[0]:Ix[-1]+1]`, so this slice is the quantity to be returned from a solver unction. A complete implementation appears in the program `wave1D_n0_ghost.y`[12].

Warning.

We have to be careful with how the spatial and temporal mesh points are stored. Say we let `x` be the physical mesh points,

```
x = linspace(0, L, Nx+1)
```

"Standard coding" of the initial condition,

```
for i in Ix:
    u_1[i] = I(x[i])
```

becomes wrong, since `u_1` and `x` have different lengths and the index `i` corresponds to two different mesh points. In fact, `x[i]` corresponds to `u[1+i]`. A correct implementation is

```
for i in Ix:
    u_1[i] = I(x[i-Ix[0]])
```

Similarly, a source term usually coded as `f(x[i], t[n])` is incorrect if `x` is defined to be the physical points, so `x[i]` must be replaced by `x[i-Ix[0]]`.

An alternative remedy is to let `x` also cover the ghost points such that `u[i]` is the value at `x[i]`.

The ghost cell is only added to the boundary where we have a Neumann ondition. Suppose we have a Dirichlet condition at $x = L$ and a homogeneous eumann condition at $x = 0$. One ghost cell $[-\Delta x, 0]$ is added to the mesh, o the index set for the physical points becomes $\{1, \ldots, N_x + 1\}$. A relevant nplementation is

```
u = zeros(Nx+2)
Ix = range(1, u.shape[0])
...
for i in Ix[:-1]:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
           dt2*f(x[i-Ix[0]], t[n])
```

[12] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0_ghost.py`


```
i = Ix[-1]
u[i] = U_0       # set Dirichlet value
i = Ix[0]
u[i-1] = u[i+1]  # update ghost value
```

The physical solution to be plotted is now in `u[1:]` or (as always) `u[Ix[0]`

7 Generalization: variable wave velocity

Our next generalization of the 1D wave equation (1) or (17) is to allc variable wave velocity c: $c = c(x)$, usually motivated by wave motion in a composed of different physical media with different properties for prop waves and hence different wave velocities c. Figure

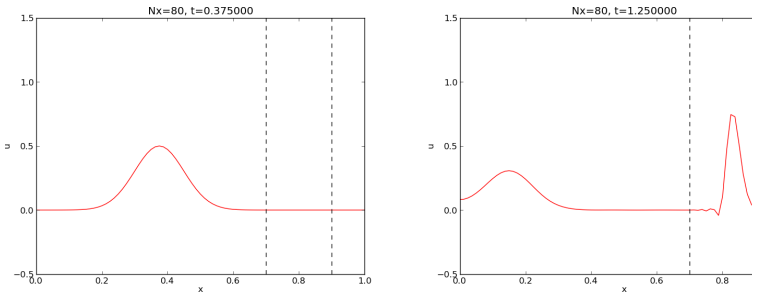


Figure 5: Left: wave entering another medium; right: transmitted and r wave .

7.1 The model PDE with a variable coefficient

Instead of working with the squared quantity $c^2(x)$ we shall for no convenience introduce $q(x) = c^2(x)$. A 1D wave equation with variab velocity often takes the form

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,.$$

This equation sampled at a mesh point (x_i, t_n) reads

$$\frac{\partial^2}{\partial t^2}u(x_i,t_n) = \frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) + f(x_i,t_n),$$

where the only new term is

$$\frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) = \left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n .$$

.2 Discretizing the variable coefficient

he principal idea is to first discretize the outer derivative. Define

$$\phi = q(x)\frac{\partial u}{\partial x},$$

nd use a centered derivative around $x = x_i$ for the derivative of ϕ:

$$\left[\frac{\partial \phi}{\partial x}\right]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x\phi]_i^n .$$

hen discretize

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}}\frac{u_{i+1}^n - u_i^n}{\Delta x} = [qD_xu]_{i+\frac{1}{2}}^n .$$

imilarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}}\frac{u_i^n - u_{i-1}^n}{\Delta x} = [qD_xu]_{i-\frac{1}{2}}^n .$$

hese intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right) . \quad (42)$$

/ith operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_xqD_xu]_i^n . \quad (43)$$

Remark.

Many are tempted to use the chain rule on the term $\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)$, but this is not a good idea when discretizing such a term.

.3 Computing the coefficient between mesh points

' q is a known function of x, we can easily evaluate $q_{i+\frac{1}{2}}$ simply as $q(x_{i+\frac{1}{2}})$ with $_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$. However, in many cases c, and hence q, is only known as a iscrete function, often at the mesh points x_i. Evaluating q between two mesh oints x_i and x_{i+1} can then be done by averaging in three ways:

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}\left(q_i + q_{i+1}\right) = [\overline{q}^x]_i, \quad \text{(arithmetic mean)} \quad (44)$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1}, \quad \text{(harmonic mean)} \quad (45)$$

$$q_{i+\frac{1}{2}} \approx \left(q_iq_{i+1}\right)^{1/2}, \quad \text{(geometric mean)} \quad (46)$$

The arithmetic mean in (44) is by far the most commonly used av technique.

With the operator notation from (44) we can specify the discretiz the complete variable-coefficient wave equation in a compact way:

$$[D_tD_tu = D_x\overline{q}^xD_xu + f]_i^n .$$

From this notation we immediately see what kind of differences that each approximated with. The notation $\overline{q}^x$ also specifies that the variable coefl approximated by an arithmetic mean, the definition being $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i +$ With the notation $[D_xqD_xu]_i^n$, we specify that q is evaluated direct function, between the mesh points: $q(x_{i-\frac{1}{2}})$ and $q(x_{i+\frac{1}{2}})$.

Before any implementation, it remains to solve (47) with respect to

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2\left(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\right) \Delta t^2 f_i^n .$$

7.4 How a variable coefficient affects the stability

The stability criterion derived in Section 10.3 reads $\Delta t \leq \Delta x/c$. If c the criterion will depend on the spatial location. We must therefore c Δt that is small enough such that no mesh cell has $\Delta x/c(x) > \Delta t$. Tha must use the largest c value in the criterion:

$$\Delta t \leq \beta\frac{\Delta x}{\max_{x\in[0,L]} c(x)} .$$

The parameter β is included as a safety factor: in some problems significantly varying c it turns out that one must choose $\beta < 1$ to hav solutions ($\beta = 0.9$ may act as an all-round value).

7.5 Neumann condition and a variable coefficient

Consider a Neumann condition $\partial u/\partial x = 0$ at $x = L = N_x\Delta x$, discretiz

$$\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0 \quad u_{i+1}^n = u_{i-1}^n,$$

for $i = N_x$. Using the scheme (48) at the end point $i = N_x$ with u_{i+1}^n results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \quad (50)$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \quad (51)$$

$$\approx -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 2q_i(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \,. \quad (52)$$

ere we used the approximation

$$q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} = q_i + \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots + q_i - \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots = 2q_i + 2\left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \mathcal{O}(\Delta x^4) \approx 2q_i \,. \quad (53)$$

An alternative derivation may apply the arithmetic mean of q in (48), leading
› the term

$$(q_i + \frac{1}{2}(q_{i+1} + q_{i-1}))(u_{i-1}^n - u_i^n) \,.$$

ince $\frac{1}{2}(q_{i+1} + q_{i-1}) = q_i + \mathcal{O}(\Delta x^2)$, we end up with $2q_i(u_{i-1}^n - u_i^n)$ for $i = N_x$
s we did above.

A common technique in implementations of $\partial u/\partial x = 0$ boundary conditions
to assume $dq/dx = 0$ as well. This implies $q_{i+1} = q_{i-1}$ and $q_{i+1/2} = q_{i-1/2}$ for
$= N_x$. The implications for the scheme are

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \quad (54)$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \,. \quad (55)$$

7.6 Implementation of variable coefficients

The implementation of the scheme with a variable wave velocity may
that c is available as an array `c[i]` at the spatial mesh points. The fo
loop is a straightforward implementation of the scheme (48):

```
for i in range(1, Nx):
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i])  - \
               0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
           dt2*f(x[i], t[n])
```

The coefficient `C2` is now defined as `(dt/dx)**2` and *not* as the squared
number since the wave velocity is variable and appears inside the parer

With Neumann conditions $u_x = 0$ at the boundary, we need to
this scheme with the discrete version of the boundary condition, as sl
Section 7.5. Nevertheless, it would be convenient to reuse the formula
interior points and just modify the indices `ip1=i+1` and `im1=i-1` as w
Section 6.3. Assuming $dq/dx = 0$ at the boundaries, we can implem
scheme at the boundary with the following code.

```
i = 0
ip1 = i+1
im1 = ip1
u[i] = - u_2[i] + 2*u_1[i] + \
       C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
       dt2*f(x[i], t[n])
```

With ghost cells we can just reuse the formula for the interior poi
at the boundary, provided that the ghost values of both u and q are c
updated to ensure $u_x = 0$ and $q_x = 0$.

A vectorized version of the scheme with a variable coefficient at
points in the mesh becomes

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

7.7 A more general model PDE with variable coeffi

Sometimes a wave PDE has a variable coefficient also in front of tl
derivative term:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t) \,.$$

A natural scheme is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n \,. \tag{57}$$

We realize that the ϱ coefficient poses no particular difficulty because the only alue ϱ_i^n enters the formula above (when written out). There is hence no need for ny averaging of ϱ. Often, ϱ will be moved to the right-hand side, also without ny difficulty:

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]_i^n \,. \tag{58}$$

.8 Generalization: damping

Waves die out by two mechanisms. In 2D and 3D the energy of the wave spreads ut in space, and energy conservation then requires the amplitude to decrease. his effect is not present in 1D. Damping is another cause of amplitude reduction. or example, the vibrations of a string die out because of damping due to air esistance and non-elastic effects in the string.

The simplest way of including damping is to add a first-order derivative to ne equation (in the same way as friction forces enter a vibrating mechanical ystem):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{59}$$

here $b \geq 0$ is a prescribed damping coefficient.

A typical discretization of (59) in terms of centered differences reads

$$[D_t D_t u + bD_{2t} u = c^2 D_x D_x u + f]_i^n \,. \tag{60}$$

Writing out the equation and solving for the unknown u_i^{n+1} gives the scheme

$$_i^{n+1} = (1+\frac{1}{2}b\Delta t)^{-1}((\frac{1}{2}b\Delta t - 1)u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t^2 f_i^n), \tag{61}$$

or $i \in \mathcal{I}_x^i$ and $n \geq 1$. New equations must be derived for u_i^1, and for boundary oints in case of Neumann conditions.

The damping is very small in many wave phenomena and then only evident or very long time simulations. This makes the standard wave equation without amping relevant for a lot of applications.

3 Building a general 1D wave equation solver

he program `wave1D_dn_vc.py`[13] is a fairly general code for 1D wave propagation roblems that targets the following initial-boundary value problem

[13] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

$$\begin{aligned}
u_t &= (c^2(x)u_x)_x + f(x,t), && x \in (0,L),\ t \in (0,T] \\
u(x,0) &= I(x), && x \in [0,L] \\
u_t(x,0) &= V(t), && x \in [0,L] \\
u(0,t) &= U_0(t) \text{ or } u_x(0,t) = 0, && t \in (0,T] \\
u(L,t) &= U_L(t) \text{ or } u_x(L,t) = 0, && t \in (0,T]
\end{aligned}$$

The `solver` function is a natural extension of the simplest `solver` fun the initial `wave1D_u0.py` program, extended with Neumann boundary co $(u_x = 0)$, a possibly time-varying boundary condition on u ($U_0(t)$, $U_L($ a variable wave velocity. The different code segments needed to mak extensions are shown and commented upon in the preceding text.

The vectorization is only applied inside the time loop, not for th condition or the first time steps, since this initial work is negligible for lo simulations in 1D problems.

The following sections explain various more advanced programming tec applied in the general 1D wave equation solver.

8.1 User action function as a class

A useful feature in the `wave1D_dn_vc.py` program is the specificatio `user_action` function as a class. Although the `plot_u` function in function of previous `wave1D*.py` programs remembers the local variable `viz` function, it is a cleaner solution to store the needed variables togeth the function, which is exactly what a class offers.

A class for flexible plotting, cleaning up files, and making a movie f function `viz` and `plot_u` did can be coded as follows:

```
class PlotSolution:
    """
    Class for the user_action function in solver.
    Visualizes the solution only.
    """
    def __init__(self,
                 casename='tmp',    # Prefix in filenames
                 umin=-1, umax=1,   # Fixed range of y axis
                 pause_between_frames=None,  # Movie speed
                 backend='matplotlib',       # or 'gnuplot'
                 screen_movie=True, # Show movie on screen?
                 title='',          # Extra message in title
                 every_frame=1):    # Show every_frame frame
        self.casename = casename
        self.yaxis = [umin, umax]
        self.pause = pause_between_frames
        module = 'scitools.easyviz.' + backend + '_'
        exec('import %s as plt' % module)
        self.plt = plt
        self.screen_movie = screen_movie
        self.title = title
        self.every_frame = every_frame
```

```
        # Clean up old movie frames
        for filename in glob('frame_*.png'):
            os.remove(filename)

    def __call__(self, u, x, t, n):
        if n % self.every_frame != 0:
            return
        title = 't=%.3g' % t[n]
        if self.title:
            title = self.title + ' ' + title
        self.plt.plot(x, u, 'r-',
                      xlabel='x', ylabel='u',
                      axis=[x[0], x[-1],
                            self.yaxis[0], self.yaxis[1]],
                      title=title,
                      show=self.screen_movie)
        # pause
        if t[n] == 0:
            time.sleep(2)  # let initial condition stay 2 s
        else:
            if self.pause is None:
                pause = 0.2 if u.size < 100 else 0
            time.sleep(pause)

        self.plt.savefig('%s_frame_%04d.png' % (self.casename, n))
```

nderstanding this class requires quite some familiarity with Python in general nd class programming in particular.

The constructor shows how we can flexibly import the plotting engine as ypically) `scitools.easyviz.gnuplot_` or `scitools.easyviz.matplotlib_` ıote the trailing underscore). With the `screen_movie` parameter we can ıppress displaying each movie frame on the screen. Alternatively, for slow ıovies associated with fine meshes, one can set `every_frame` to, e.g., 10, causing very 10 frames to be shown.

The `__call__` method makes `PlotSolution` instances behave like functions, o we can just pass an instance, say `p`, as the `user_action` argument in the `olver` function, and any call to `user_action` will be a call to `p.__call__`.

.2 Pulse propagation in two media

he function `pulse` in `wave1D_dn_vc.py` demonstrates wave motion in heteroge- eous media where c varies. One can specify an interval where the wave velocity decreased by a factor `slowness_factor` (or increased by making this factor ss than one). Four types of initial conditions are available: a rectangular ulse (`plug`), a Gaussian function (`gaussian`), a "cosine hat" consisting of one eriod of the cosine function (`cosinehat`), and half a period of a "cosine hat" `alf-cosinehat`). These peak-shaped initial conditions can be placed in the ıiddle (`loc='center'`) or at the left end (`loc='left'`) of the domain. The `ulse` function is a flexible tool for playing around with various wave shapes nd location of a medium with a different wave velocity:


```
def pulse(C=1, Nx=200, animate=True, version='vectorized', T=2,
          loc='center', pulse_tp='gaussian', slowness_factor=2,
          medium=[0.7, 0.9], every_frame=1, sigma=0.05):
    """
    Various peaked-shaped initial conditions on [0,1].
    Wave velocity is decreased by the slowness_factor inside
    medium. The loc parameter can be 'center' or 'left',
    depending on where the initial pulse is to be located.
    The sigma parameter governs the width of the pulse.
    """
    # Use scaled parameters: L=1 for domain length, c_0=1
    # for wave velocity outside the domain.
    L = 1.0
    c_0 = 1.0
    if loc == 'center':
        xc = L/2
    elif loc == 'left':
        xc = 0

    if pulse_tp in ('gaussian','Gaussian'):
        def I(x):
            return exp(-0.5*((x-xc)/sigma)**2)
    elif pulse_tp == 'plug':
        def I(x):
            return 0 if abs(x-xc) > sigma else 1
    elif pulse_tp == 'cosinehat':
        def I(x):
            # One period of a cosine
            w = 2
            a = w*sigma
            return 0.5*(1 + cos(pi*(x-xc)/a)) \
                   if xc - a <= x <= xc + a else 0

    elif pulse_tp == 'half-cosinehat':
        def I(x):
            # Half a period of a cosine
            w = 4
            a = w*sigma
            return cos(pi*(x-xc)/a) \
                   if xc - 0.5*a <= x <= xc + 0.5*a else 0
    else:
        raise ValueError('Wrong pulse_tp="%s"' % pulse_tp)

    def c(x):
        return c_0/slowness_factor \
               if medium[0] <= x <= medium[1] else c_0

    umin=-0.5; umax=1.5*I(xc)
    casename = '%s_Nx%s_sf%s' % \
               (pulse_tp, Nx, slowness_factor)
    action = PlotMediumAndSolution(
        medium, casename=casename, umin=umin, umax=umax,
        every_frame=every_frame, screen_movie=animate)

    dt = (L/Nx)/c  # choose the stability limit with given Nx
    # Lower C will then use this dt, but smaller Nx
    solver(I=I, V=None, f=None, c=c, U_0=None, U_L=None,
           L=L, dt=dt, C=C, T=T,
           user_action=action, version=version,
           stability_safety_factor=1)
```

he `PlotMediumAndSolution` class used here is a subclass of `PlotSolution` here the medium with reduced c value, as specified by the `medium` interval, is isualized in the plots.

Notice.

The argument N_x in the `pulse` function does not correspond to the actual spatial resolution of $C < 1$, since the `solver` function takes a fixed Δt and C, and adjusts Δx accordingly. As seen in the `pulse` function, the specified Δt is chosen according to the limit $C = 1$, so if $C < 1$, Δt remains the same, but the `solver` function operates with a larger Δx and smaller N_x than was specified in the call to `pulse`. The practical reason is that we always want to keep Δt fixed such that plot frames and movies are synchronized in time regardless of the value of C (i.e., Δx is varies when the Courant number varies).

The reader is encouraged to play around with the `pulse` function:

```
>>> import wave1D_dn_vc as w
>>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, every_frame=10)
```

o easily kill the graphics by Ctrl-C and restart a new simulation it might be asier to run the above two statements from the command line with

```
erminal> python -c 'import wave1D_dn_vc as w; w.pulse(...)'
```

Exercises

Exercise 6: Find the analytical solution to a damped wave equation

Consider the wave equation with damping (59). The goal is to find an exact olution to a wave problem with damping. A starting point is the standing wave olution from Exercise 1. It becomes necessary to include a damping term e^{-ct} nd also have both a sine and cosine component in time:

$$u_{\mathrm{e}}(x,t) = e^{-\beta t}\sin kx\left(A\cos\omega t + B\sin\omega t\right)\ .$$

ind k from the boundary conditions $u(0,t) = u(L,t) = 0$. Then use the PDE o find constraints on β, ω, A, and B. Set up a complete initial-boundary value roblem and its solution. Filename: `damped_waves.pdf`.

Problem 7: Explore symmetry boundary conditions

Consider the simple "plug" wave where $\Omega = [-L, L]$ and

$$I(x) = \begin{cases} 1, & x \in [-\delta, \delta], \\ 0, & \text{otherwise} \end{cases}$$

for some number $0 < \delta < L$. The other initial condition is $u_t(x,0) = 0$ a is no source term f. The boundary conditions can be set to $u = 0$. The to this problem is symmetric around $x = 0$. This means that we can s the wave process in only the half of the domain $[0, L]$.

a) Argue why the symmetry boundary condition is $u_x = 0$ at $x = 0$.

Hint. Symmetry of a function about $x = x_0$ means that $f(x_0 + h) = f$

b) Perform simulations of the complete wave problem from on $[-L, L]$ after, utilize the symmetry of the solution and run a simulation in ha domain $[0, L]$, using a boundary condition at $x = 0$. Compare the two s and make sure that they are the same.

c) Prove the symmetry property of the solution by setting up the c initial-boundary value problem and showing that if $u(x,t)$ is a solutic also $u(-x,t)$ is a solution.
Filename: `wave1D_symmetric`.

Exercise 8: Send pulse waves through a layered medi

Use the `pulse` function in `wave1D_dn_vc.py` to investigate sending located with its peak at $x = 0$, through the medium to the right wher another medium for $x \in [0.7, 0.9]$ where the wave velocity is decreas factor s_f. Report what happens with a Gaussian pulse, a "cosine hat half a "cosine hat" pulse, and a plug pulse for resolutions $N_x = 40, 80, 1$ $s_f = 2, 4$. Use $C = 1$ in the medium outside $[0.7, 0.9]$. Simulate unti
Filename: `pulse1D.py`.

Exercise 9: Compare discretizations of a Neumann tion

We have a 1D wave equation with variable wave velocity: $u_t = (qu$ Neumann condition u_x at $x = 0, L$ can be discretized as shown in (52) a

The aim of this exercise is to examine the rate of the numerical err using different ways of discretizing the Neumann condition. As test p $q = 1 + (x - L/2)^4$ can be used, with $f(x,t)$ adapted such that the solu a simple form, say $u(x,t) = \cos(\pi x/L)\cos(\omega t)$ for some $\omega = \sqrt{q}\pi/L$.

a) Perform numerical experiments and find the convergence rate of t using the approximation and (55).

) Switch to $q(x) = \cos(\pi x/L)$, which is symmetric at $x = 0, L$, and check the onvergence rate of the scheme (55). Now, $q_{i-1/2}$ is a 2nd-order approximation to , $q_{i-1/2} = q_i + 0.25q_i''\Delta x^2 + \cdots$, because $q_i' = 0$ for $i = N_x$ (a similar argument an be applied to the case $i = 0$).

) A third discretization can be based on a simple and convenient, but less ccurate, one-sided difference: $u_i - u_{i-1} = 0$ at $i = N_x$ and $u_{i+1} - u_i = 0$ at $= 0$. Derive the resulting scheme in detail and implement it. Run experiments o establish the rate of convergence.

) A fourth technique is to view the scheme as

$$[D_tD_tu]_i^n = \frac{1}{\Delta x}\left([qD_xu]_{i+\frac{1}{2}}^n - [qD_xu]_{i-\frac{1}{2}}^n\right) + [f]_i^n,$$

nd place the boundary at $x_{i+\frac{1}{2}}$, $i = N_x$, instead of exactly at the physical oundary. With this idea, we can just set $[qD_xu]_{i+\frac{1}{2}}^n = 0$. Derive the complete cheme using this technique. The implementation of the boundary condition at $- \Delta x/2$ is $\mathcal{O}(\Delta x^2)$ accurate, but the interesting question is what impact the ıovement of the boundary has on the convergence rate (compute the errors as sual over the entire mesh).

Exercise 10: Verification by a cubic polynomial in space

he purpose of this exercise is to verify the implementation of the `solver` nction in the program `wave1D_n0.py`[14] by using an exact numerical solution or the wave equation $u_{tt} = c^2u_{xx} + f$ with Neumann boundary conditions $_x(0,t) = u_x(L,t) = 0$.

A similar verification is used in the file `wave1D_u0.py`[15], which solves the ame PDE, but with Dirichlet boundary conditions $u(0,t) = u(L,t) = 0$. The lea of the verification test in function `test_quadratic` in `wave1D_u0.py` is to solution that is a lower-order polynomial such that both the PDE problem, the oundary conditions, and all the discrete equations are exactly fulfilled. Then ıe `solver` function should reproduce this exact solution to machine precision. Iore precisely, we seek $u = X(x)T(t)$, with $T(t)$ as a linear function and $X(x)$ s a parabola that fulfills the boundary conditions. Inserting this u in the PDE etermines f. It tuns out that u also fulfills the discrete equations, because the uncation error of the discretized PDE has derivatives in x and t of order four nd higher. These derivatives all vanish for a quadratic $X(x)$ and linear $T(t)$.

It would be attractive to use a similar approach in the case of Neumann onditions. We set $u = X(x)T(t)$ and seek lower-order polynomials X and T. o force u_x to vanish at the boundary, we let X_x be a parabola. Then X is a ıbic polynomial. The fourth-order derivative of a cubic polynomial vanishes, so $= X(x)T(t)$ will fulfill the discretized PDE also in this case, if f is adjusted ıch that u fulfills the PDE.

[14] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0.py`
[15] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0.py`

However, the discrete boundary condition is not exactly fulfilled choice of u. The reason is that

$$[D_{2x}u]_i^n = u_x(x_i,t_n) + \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

At the boundary two boundary points, $X_x(x) = 0$ such that $u_x = 0$. H u_{xxx} is a constant and not zero when $X(x)$ is a cubic polynomial. Th our $u = X(x)T(t)$ fulfills

$$[D_{2x}u]_i^n = \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2,$$

and not

$$[D_{2x}u]_i^n = 0, quadi = 0, N_x,$$

as it should. (Note that all the higher-order terms $\mathcal{O}(\Delta x^4)$ also have order derivatives that vanish for a cubic polynomial.) So to summar fundamental problem is that u as a product of a cubic polynomial and or quadratic polynomial in time is not an exact solution of the discrete b conditions.

To make progress, we assume that $u = X(x)T(t)$, where T for simp taken as a prescribed linear function $1 + \frac{1}{2}t$, and $X(x)$ is taken as an u cubic polynomial $\sum_{j=0}^3 a_jx^j$. There are two different ways of determi coefficients $a_0, \ldots, a_3$ such that both the discretized PDE and the dis boundary conditions are fulfilled, under the constraint that we can s function $f(x,t)$ for the PDE to feed to the `solver` function in `wave1D` Both approaches are explained in the subexercises.

a) One can insert u in the discretized PDE and find the corresponding one can insert u in the discretized boundary conditions. This yields two e for the four coefficients $a_0, \ldots, a_3$. To find the coefficients, one can se and $a_1 = 1$ for simplicity and then determine a_2 and a_3. This appro make a_2 and a_3 depend on Δx and f will depend on both Δx and Δt.

Use `sympy` to perform analytical computations. A starting point is t u as follows:

```
def test_cubic1():
    import sympy as sm
    x, t, c, L, dx, dt = sm.symbols('x t c L dx dt')
    i, n = sm.symbols('i n', integer=True)

    # Assume discrete solution is a polynomial of degree 3 in x
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    a = sm.symbols('a_0 a_1 a_2 a_3')
    X = lambda x: sum(a[q]*x**q for q in range(4))  # Spatial ter
    u = lambda x, t: X(x)*T(t)
```

The symbolic expression for u is reached by calling `u(x,t)` with `x` a `sympy` symbols.

Define `DxDx(u, i, n)`, `DtDt(u, i, n)`, and `D2x(u, i, n)` as Python functions for returning the difference approximations $[D_xD_xu]_i^n$, $[D_tD_tu]_i^n$, and $[D_{2x}u]_i^n$. The next step is to set up the residuals for the equations $[D_{2x}u]_0^n = 0$ and $[D_{2x}u]_{N_x}^n = 0$, where $N_x = L/\Delta x$. Call the residuals `R_0` and `R_L`. Substitute a_0 and a_1 by 0 and 1, respectively, in `R_0`, `R_L`, and `a`:

```
R_0 = R_0.subs(a[0], 0).subs(a[1], 1)
R_L = R_L.subs(a[0], 0).subs(a[1], 1)
a = list(a)  # enable in-place assignment
a[0:2] = 0, 1
```

Determining a_2 and a_3 from the discretized boundary conditions is then about solving two equations with respect to a_2 and a_3, i.e., `a[2:]`:

```
s = sm.solve([R_0, R_L], a[2:])
# s is dictionary with the unknowns a[2] and a[3] as keys
a[2:] = s[a[2]], s[a[3]]
```

Now, `a` contains computed values and `u` will automatically use these new values since `X` accesses `a`.

Compute the source term f from the discretized PDE: $f_i^n = [D_tD_tu - c^2D_xD_xu]_i^n$. Turn u, the time derivative u_t (needed for the initial condition $V(x)$), and f into Python functions. Set numerical values for L, N_x, C, and c. Prescribe the time interval as $\Delta t = CL/(N_xc)$, which imply $\Delta x = c\Delta t/C = L/N_x$. Define new functions `I(x)`, `V(x)`, and `f(x,t)` as wrappers of the ones made above, where fixed values of L, c, Δx, and Δt are inserted, such that `I`, `V`, and `f` can be passed on to the `solver` function. Finally, call `solver` with a `user_action` function that compares the numerical solution to this exact solution u of the discrete PDE problem.

Hint. To turn a `sympy` expression `e`, depending on a series of symbols, say `x`, `t`, `dx`, `dt`, `L`, and `c`, into plain Python function `e_exact(x,t,L,dx,dt,c)`, one can write

```
e_exact = sm.lambdify([x,t,L,dx,dt,c], e, 'numpy')
```

The `'numpy'` argument is a good habit as the `e_exact` function will then work with array arguments if it contains mathematical functions (but here we only do plain arithmetics, which automatically work with arrays).

) An alternative way of determining $a_0, \ldots, a_3$ is to reason as follows. We first construct $X(x)$ such that the boundary conditions are fulfilled: $X = x(L-x)$. However, to compensate for the fact that this choice of X does not fulfill the discrete boundary condition, we seek u such that

$$u_x = \frac{\partial}{\partial x}x(L-x)T(t) - \frac{1}{6}u_{xxx}\Delta x^2,$$

since this u will fit the discrete boundary condition. Assuming $u = T(t)\sum$ we can use the above equation to determine the coefficients a_1, a_2, a_3. e.g., 1 can be used for a_0. The following `sumpy` code computes this u:

```
def test_cubic2():
    import sympy as sm
    x, t, c, L, dx = sm.symbols('x t c L dx')
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    # Set u as a 3rd-degree polynomial in space
    X = lambda x: sum(a[i]*x**i for i in range(4))
    a = sm.symbols('a_0 a_1 a_2 a_3')
    u = lambda x, t: X(x)*T(t)
    # Force discrete boundary condition to be zero by adding
    # a correction term the analytical suggestion x*(L-x)*T
    # u_x = x*(L-x)*T(t) - 1/6*u_xxx*dx**2
    R = sm.diff(u(x,t), x) - (
        x*(L-x) - sm.Rational(1,6)*sm.diff(u(x,t), x, x, x)*dx**2
    # R is a polynomial: force all coefficients to vanish.
    # Turn R to Poly to extract coefficients:
    R = sm.poly(R, x)
    coeff = R.all_coeffs()
    s = sm.solve(coeff, a[1:])  # a[0] is not present in R
    # s is dictionary with a[i] as keys
    # Fix a[0] as 1
    s[a[0]] = 1
    X = lambda x: sm.simplify(sum(s[a[i]]*x**i for i in range(4))
    u = lambda x, t: X(x)*T(t)
    print 'u:', u(x,t)
```

The next step is to find the source term `f_e` by inserting `u_e` in tl Thereafter, turn `u`, `f`, and the time derivative of `u` into plain Python fu as in a), and then wrap these functions in new functions `I`, `V`, and `f`, v right signature as required by the `solver` function. Set parameters as ir check that the solution is exact to machine precision at each time level u appropriate `user_action` function.
Filename: `wave1D_n0_test_cubic.py`.

10 Analysis of the difference equations

10.1 Properties of the solution of the wave equatio

The wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}$$

has solutions of the form

$$u(x,t) = g_R(x-ct) + g_L(x+ct),$$

for any functions g_R and g_L sufficiently smooth to be differentiated twi result follows from inserting (68) in the wave equation. A function of t $g_R(x-ct)$ represents a signal moving to the right in time with constant

This feature can be explained as follows. At time $t = 0$ the signal looks like $_R(x)$. Introducing a moving x axis with coordinates $\xi = x - ct$, we see the unction $g_R(\xi)$ is "at rest" in the ξ coordinate system, and the shape is always the ame. Say the $g_R(\xi)$ function has a peak at $\xi = 0$. This peak is located at $x = ct$, hich means that it moves with the velocity $dx/dt = c$ in the x coordinate ystem. Similarly, $g_L(x + ct)$ is a function initially with shape $g_L(x)$ that moves n the negative x direction with constant velocity c (introduce $\xi = x + ct$, look t the point $\xi = 0$, $x = -ct$, which has velocity $dx/dt = -c$).

With the particular initial conditions

$$u(x, 0) = I(x), \quad \frac{\partial}{\partial t}u(x, 0) = 0,$$

e get, with u as in (68),

$$g_R(x) + g_L(x) = I(x), \quad -cg'_R(x) + cg'_L(x) = 0,$$

hich have the solution $g_R = g_L = I/2$, and consequently

$$u(x, t) = \frac{1}{2}I(x - ct) + \frac{1}{2}I(x + ct)\,. \tag{69}$$

he interpretation of (69) is that the initial shape of u is split into two parts, ach with the same shape as I but half of the initial amplitude. One part is aveling to the left and the other one to the right.

The solution has two important physical features: constant amplitude of the ft and right wave, and constant velocity of these two waves. It turns out that ne numerical solution will also preserve the constant amplitude, but the velocity epends on the mesh parameters Δt and Δx.

The solution (69) will be influenced by boundary conditions when the parts $I(x - ct)$ and $\frac{1}{2}I(x + ct)$ hit the boundaries and get, e.g., reflected back into ne domain. However, when $I(x)$ is nonzero only in a small part in the middle of ne spatial domain $[0, L]$, which means that the boundaries are placed far away om the initial disturbance of u, the solution (69) is very clearly observed in a mulation.

A useful representation of solutions of wave equations is a linear combination f sine and/or cosine waves. Such a sum of waves is a solution if the governing DE is linear and each sine or cosine wave fulfills the equation. To ease analytical alculations by hand we shall work with complex exponential functions instead f real-valued sine or cosine functions. The real part of complex expressions ill typically be taken as the physical relevant quantity (whenever a physical elevant quantity is strictly needed). The idea now is to build $I(x)$ of complex ave components e^{ikx}:

$$I(x) \approx \sum_{k\in K} b_k e^{ikx}\,. \tag{70}$$

ere, k is the frequency of a component, K is some set of all the discrete k values eeded to approximate $I(x)$ well, and b_k are constants that must be determined. /e will very seldom need to compute the b_k coefficients: most of the insight we

look for and the understanding of the numerical methods we want to e come from investigating how the PDE and the scheme treat a single co e^{ikx} wave.

Letting the number of k values in K tend to infinity makes the s converge to $I(x)$, and this sum is known as a *Fourier series* represent $I(x)$. Looking at (69), we see that the solution $u(x, t)$, when $I(x)$ is rep as in (70), is also built of basic complex exponential wave component form $e^{ik(x\pm ct)}$ according to

$$u(x, t) = \frac{1}{2}\sum_{k\in K} b_k e^{ik(x-ct)} + \frac{1}{2}\sum_{k\in K} b_k e^{ik(x+ct)}\,.$$

It is common to introduce the frequency in time $\omega = kc$ and assume tha is a sum of basic wave components written as $e^{ikx-\omega t}$. (Observe that i such a wave component in the governing PDE reveals that $\omega^2 = k^2c^2$, o reflecting the two solutions: one ($+kc$) traveling to the right and the oth traveling to the left.)

10.2 More precise definition of Fourier representati

The quick intuitive introduction above to representing a function by a sine and cosine waves suffices as background for the forthcoming mat analyzing a single wave component. However, to understand all details different wave components sum up to the analytical and numerical sol more precise mathematical treatment is helpful and therefore summarize

It is well known that periodic functions can be represented by Fouri A generalization of the Fourier series idea to non-periodic functions de the real line is the *Fourier transform*:

$$I(x) = \int_{-\infty}^{\infty} A(k)e^{ikx}dk,$$
$$A(k) = \int_{-\infty}^{\infty} I(x)e^{-ikx}dx\,.$$

The function $A(k)$ reflects the weight of each wave component e^{ikx} in an sum of such wave components. That is, $A(k)$ reflects the frequency co the function $I(x)$. Fourier transforms are particularly fundamental for a and understanding time-varying signals.

The solution of the linear 1D wave PDE can be expressed as

$$u(x, t) = \int_{-\infty}^{\infty} A(k)e^{i(kx-\omega(k)t)}dx\,.$$

In a finite difference method, we represent u by a mesh function u^n_q, counts temporal mesh points and q counts the spatial ones (the usual for spatial points, i, is here already used as imaginary unit). Simila

approximated by the mesh function I_q, $q = 0, \ldots, N_x$. On a mesh, it does ot make sense to work with wave components e^{ikx} for very large k, because ie shortest possible sine or cosine wave that can be represented on a mesh ith spacing Δx is the wave with wavelength $2\Delta x$ (the sine/cosine signal mps up and down between each mesh point). The corresponding k value is $= 2\pi/(2\Delta x) = \pi/\Delta x$, known as the *Nyquist frequency*. Within the range of elevant frequencies $(0, \pi/\Delta x]$ one defines the discrete Fourier transform[16], using $N_x + 1$ discrete frequencies:

$$I_q = \frac{1}{N_x+1}\sum_{k=0}^{N_x} A_k e^{i2\pi kj/(N_x+1)}, \quad i = 0, \ldots, N_x, \tag{74}$$

$$A_k = \sum_{q=0}^{N_x} I_q e^{-i2\pi kq/(N_x+1)}, \quad k = 0, \ldots, N_x + 1\,. \tag{75}$$

he A_k values is the discrete Fourier transform of the I_q values, and the latter re the inverse discrete Fourier transform of the A_k values.

The discrete Fourier transform is efficiently computed by the *Fast Fourier ansform* algorithm. For a real function $I(x)$ the relevant Python code for omputing and plotting the discrete Fourier transform appears in the example elow.

```
import numpy as np
from numpy import sin

def I(x):
    return sin(2*pi*x) + 0.5*sin(4*pi*x) + 0.1*sin(6*pi*x)

# Mesh
L = 10; Nx = 100
x = np.linspace(0, L, Nx+1)
dx = L/float(Nx)

# Discrete Fourier transform
A = np.fft.rfft(I(x))
A_amplitude = np.abs(A)

# Compute the corresponding frequencies
freqs = np.linspace(0, pi/dx, A_amplitude.size)

import matplotlib.pyplot as plt
plt.plot(freqs, A_amplitude)
plt.show()
```

0.3 Stability

he scheme

[16] http://en.wikipedia.org/wiki/Discrete_Fourier_transform

$$[D_t D_t u = c^2 D_x D_x u]_q^n$$

for the wave equation $u_t = c^2 u_{xx}$ allows basic wave components

$$u_q^n = e^{i(kx_q - \tilde{\omega} t_n)}$$

as solution, but it turns out that the frequency in time, $\tilde{\omega}$, is not equa exact $\omega = kc$. The idea now is to study how the scheme treats an arbitra component with a given k. We ask two key questions:

- How accurate is $\tilde{\omega}$ compared to ω?
- Does the amplitude of such a wave component preserve its (unit) an as it should, or does it get amplified or damped in time (due to a $\tilde{\omega}$)?

The following analysis will answer these questions. Note the need for us counter for the mesh point in x direction since i is already used as the in unit (in this analysis).

Preliminary results. A key result needed in the investigations is tl difference approximation of a second-order derivative acting on a compl component:

$$[D_t D_t e^{i\omega t}]^n = -\frac{4}{\Delta t^2}\sin^2\left(\frac{\omega\Delta t}{2}\right) e^{i\omega n\Delta t}\,.$$

By just changing symbols ($\omega \to k$, $t \to x$, $n \to q$) it follows that

$$[D_x D_x e^{ikx}]_q = -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x}\,.$$

Numerical wave propagation. Inserting a basic wave componen $e^{i(kx_q - \tilde{\omega} t_n)}$ in (76) results in the need to evaluate two expressions:

$$\begin{aligned}
[D_t D_t e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_t D_t e^{-i\tilde{\omega}t}]^n e^{ikq\Delta x} \\
&= -\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) e^{-i\tilde{\omega}n\Delta t} e^{ikq\Delta x} \\
[D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_x D_x e^{ikx}]_q e^{-i\tilde{\omega}n\Delta t} \\
&= -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x} e^{-i\tilde{\omega}n\Delta t}\,.
\end{aligned}$$

Then the complete scheme,

$$[D_t D_t e^{ikx} e^{-i\tilde{\omega}t} = c^2 D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n$$

:ads to the following equation for the unknown numerical frequency $\tilde{\omega}$ (after ividing by $-e^{ikx}e^{-i\tilde{\omega}t}$):

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right),$$

r

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2\sin^2\left(\frac{k\Delta x}{2}\right), \tag{79}$$

here

$$C = \frac{c\Delta t}{\Delta x} \tag{80}$$

the Courant number. Taking the square root of (79) yields

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C\sin\left(\frac{k\Delta x}{2}\right), \tag{81}$$

ince the exact ω is real it is reasonable to look for a real solution $\tilde{\omega}$ of (81). he right-hand side of (81) must then be in $[-1,1]$ because the sine function n the left-hand side has values in $[-1,1]$ for real $\tilde{\omega}$. The sine function on the ght-hand side can attain the value 1 when

$$\frac{k\Delta x}{2} = m\frac{\pi}{2}, \quad m \in \mathbb{Z}.$$

/ith $m = 1$ we have $k\Delta x = \pi$, which means that the wavelength $\lambda = 2\pi/k$ ecomes $2\Delta x$. This is the absolutely shortest wavelength that can be represented n the mesh: the wave jumps up and down between each mesh point. Larger alues of $|m|$ are irrelevant since these correspond to k values whose waves are oo short to be represented on a mesh with spacing Δx. For the shortest possible ave in the mesh, $\sin(k\Delta x/2) = 1$, and we must require

$$C \leq 1. \tag{82}$$

Consider a right-hand side in (81) of magnitude larger than unity. The olution $\tilde{\omega}$ of (81) must then be a complex number $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$ because the ne function is larger than unity for a complex argument. One can show that or any ω_i there will also be a corresponding solution with $-\omega_i$. The component ith $\omega_i > 0$ gives an amplification factor $e^{\omega_i t}$ that grows exponentially in time. /e cannot allow this and must therefore require $C \leq 1$ as a *stability criterion*.

Remark.

For smoother wave components with longer wave lengths per length Δx, (82) can in theory be relaxed. However, small round-off errors are always present in a numerical solution and these vary arbitrarily from mesh point to mesh point and can be viewed as unavoidable noise with wavelength

$2\Delta x$. As explained, $C > 1$ will for this very small noise lead to expone growth of the shortest possible wave component in the mesh. This will therefore grow with time and destroy the whole solution.

10.4 Numerical dispersion relation

Equation (81) can be solved with respect to $\tilde{\omega}$:

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(C\sin\left(\frac{k\Delta x}{2}\right)\right).$$

The relation between the numerical frequency $\tilde{\omega}$ and the other paramet Δx, and Δt is called a *numerical dispersion relation*. Correspondingly, ω the *analytical dispersion relation*.

The special case $C = 1$ deserves attention since then the right-hanc (83) reduces to

$$\frac{2}{\Delta t}\frac{k\Delta x}{2} = \frac{1}{\Delta t}\frac{\omega\Delta x}{c} = \frac{\omega}{C} = \omega.$$

That is, $\tilde{\omega} = \omega$ and the numerical solution is exact at all mesh points rega Δx and Δt! This implies that the numerical solution method is also an ar solution method, at least for computing u at discrete points (the nu method says nothing about the variation of u *between* the mesh poi employing the common linear interpolation for extending the discrete s gives a curve that deviates from the exact one).

For a closer examination of the error in the numerical dispersion when $C < 1$, we can study $\tilde{\omega} - \omega$, $\tilde{\omega}/\omega$, or the similar error measures velocity: $\tilde{c} - c$ and $\tilde{c}/c$, where $c = \omega/k$ and $\tilde{c} = \tilde{\omega}/k$. It appears that t convenient expression to work with is $\tilde{c}/c$:

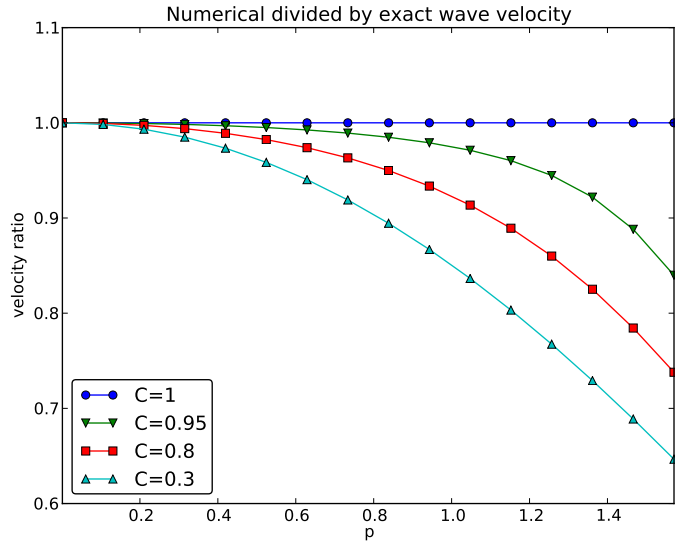
$$\frac{\tilde{c}}{c} = \frac{1}{Cp}\sin^{-1}(C\sin p),$$

with $p = k\Delta x/2$ as a non-dimensional measure of the spatial freque essence, p tells how many spatial mesh points we have per wave length of the wave component with frequency k (the wave length is $2\pi/k$). Tl reflects how well the spatial variation of the wave component is resolve mesh. Wave components with wave length less than $2\Delta x$ ($2\pi/k < 2\Delta x$) visible in the mesh, so it does not make sense to have $p > \pi/2$.

We may introduce the function $r(C,p) = \tilde{c}/c$ for further investig numerical errors in the wave velocity:

$$r(C,p) = \frac{1}{Cp}\sin^{-1}(C\sin p), \quad C \in (0,1],\ p \in (0,\pi/2].$$

This function is very well suited for plotting since it combines several par in the problem into a dependence on two non-dimensional numbers, C



igure 6: The fractional error in the wave velocity for different Courant numbers.

Defining

```
def r(C, p):
    return 2/(C*p)*asin(C*sin(p))
```

e can plot $r(C,p)$ as a function of p for various values of C, see Figure 6. Note hat the shortest waves have the most erroneous velocity, and that short waves ıove more slowly than they should.

With `sympy` we can also easily make a Taylor series expansion in the dis-:retization parameter p:

```
>>> C, p = symbols('C p')
>>> # Compute the 7 first terms around p=0 with no O() term
>>> rs = r(C, p).series(p, 0, 7).removeO()
>>> rs
**6*(5*C**6/112 - C**4/16 + 13*C**2/720 - 1/5040) +
**4*(3*C**4/40 - C**2/12 + 1/120) +
**2*(C**2/6 - 1/6) + 1
>>> rs_error_leading_order = (rs - 1).extract_leading_order(p)
>>> rs_error_leading_order
**2*(C**2/6 - 1/6)
>>> # Turn the series expansion into a Python function
>>> rs_pyfunc = lambdify([C, p], rs, modules='numpy')
>>> # Check: rs_pyfunc is exact (=1) for C=1
>>> rs_pyfunc(1, 0.1)
1.0
```


Without the `.removeO()` call the series get an `O(x**7)` term that n impossible to convert the series to a Python function (for, e.g., plotting

From the `rs_error_leading_order` expression above we see that the order term in the error of this series expansion is

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2 (C^2-1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right),$$

pointing to an error $\mathcal{O}(\Delta t^2, \Delta x^2)$, which is compatible with the error difference approximations (D_tD_t and D_xD_x).

Here is an alternative way of performing a series expansion: we `lseries` method, which returns an iterator over all the terms in the ex and ask for the 4 first terms (via `itertools.islice`, which can slice an i Collecting the terms in a list makes it possible to factor each term indi Summing up the terms results in a nicer expression:

```
>>> import itertools
>>> rs = [t for t in itertools.islice(r(C, p).lseries(p), 4)]
>>> rs
[1, C**2*p**2/6 - p**2/6,
 3*C**4*p**4/40 - C**2*p**4/12 + p**4/120,
 5*C**6*p**6/112 - C**4*p**6/16 + 13*C**2*p**6/720 - p**6/5040]
>>> rs = [factor(t) for t in rs]
>>> rs
[1, p**2*(C - 1)*(C + 1)/6,
 p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120,
 p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040]
>>> rs = sum(rs)  # Python's sum function
>>> rs
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

We see from the last expression that $C = 1$ makes all the terms in `rs` Since we already know that the numerical solution is exact for $C =$ remaining terms in the Taylor series expansion will also contain factors and cancel for $C = 1$.

10.5 Extending the analysis to 2D and 3D

The typical analytical solution of a 2D wave equation

$$u_{tt} = c^2(u_{xx} + u_{yy}),$$

is a wave traveling in the direction of $\boldsymbol{k} = k_x\boldsymbol{i} + k_y\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ vectors in the x and y directions, respectively. Such a wave can be expr

$$u(x,y,t) = g(k_x x + k_y y - kct)$$

for some twice differentiable function g, or with $\omega = kc$, $k = |\boldsymbol{k}|$:

$$u(x,y,t) = g(k_x x + k_y y - \omega t)\,.$$

/e can in particular build a solution by adding complex Fourier components of
ıe form

$$\exp\left(i(k_x x + k_y y - \omega t)\right).$$

A discrete 2D wave equation can be written as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]^n_{q,r}. \tag{86}$$

his equation admits a Fourier component

$$u^n_{q,r} = \exp\left(i(k_x q\Delta x + k_y r\Delta y - \tilde{\omega} n\Delta t)\right), \tag{87}$$

s solution. Letting the operators D_tD_t, D_xD_x, and D_yD_y act on $u^n_{q,r}$ from (87)
ansforms (86) to

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) + c^2\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right). \tag{88}$$

[

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2\sin^2 p_x + C_y^2\sin^2 p_y, \tag{89}$$

here we have eliminated the factor 4 and introduced the symbols

$$C_x = \frac{c^2\Delta t^2}{\Delta x^2}, \quad C_y = \frac{c^2\Delta t^2}{\Delta y^2}, \quad p_x = \frac{k_x\Delta x}{2}, \quad p_y = \frac{k_y\Delta y}{2}.$$

or a real-valued $\tilde{\omega}$ the right-hand side must be less than or equal to unity in
bsolute value, requiring in general that

$$C_x^2 + C_y^2 \leq 1. \tag{90}$$

his gives the stability criterion, more commonly expressed directly in an
ıequality for the time step:

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{91}$$

similar, straightforward analysis for the 3D case leads to

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{92}$$

ı the case of a variable coefficient $c^2 = c^2(\boldsymbol{x})$, we must use the worst-case value

$$\bar{c} = \sqrt{\max_{\boldsymbol{x}\in\Omega} c^2(\boldsymbol{x})} \tag{93}$$

ı the stability criteria. Often, especially in the variable wave velocity case, it is
ise to introduce a safety factor $\beta \in (0, 1]$ too:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2}$$

The exact numerical dispersion relations in 2D and 3D becomes, for c
c,

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin_y^p\right)^{\frac{1}{2}}\right),$$
$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin_y^p + C_z^2\sin_z^p\right)^{\frac{1}{2}}\right).$$

We can visualize the numerical dispersion error in 2D much like we di
To this end, we need to reduce the number of parameters in $\tilde{\omega}$. The dire
the wave is parameterized by the polar angle θ, which means that

$$k_x = k\sin\theta, \quad k_y = k\cos\theta.$$

A simplification is to set $\Delta x = \Delta y = h$. Then $C_x = C_y = c\Delta t/h$, which
C. Also,

$$p_x = \frac{1}{2}kh\cos\theta, \quad p_y = \frac{1}{2}kh\sin\theta.$$

The numerical frequency $\tilde{\omega}$ is now a function of three parameters:

- C reflecting the number cells a wave is displaced during a time st
- kh reflecting the number of cells per wave length in space
- θ expressing the direction of the wave

We want to visualize the error in the numerical frequency. To avoid ha
as a free parameter in $\tilde{\omega}$, we work with $\tilde{c}/c$, because the fraction $2/\Delta t$
rewritten as

$$\frac{2}{kc\Delta t} = \frac{2}{2kc\Delta t h/h} = \frac{1}{Ckh},$$

and

$$\frac{\tilde{c}}{c} = \frac{1}{Ckh}\sin^{-1}\left(C\left(\sin^2(\frac{1}{2}kh\cos\theta) + \sin^2(\frac{1}{2}kh\sin\theta)\right)^{\frac{1}{2}}\right).$$

We want to visualize this quantity as a function of kh and θ for some v
$C \leq 1$. It is instructive to make color contour plots of $1-\tilde{c}/c$ in *polar coo*
with θ as the angular coordinate and kh as the radial coordinate.

The stability criterion (90) becomes $C \leq C_{\max} = 1/\sqrt{2}$ in the pre
case with the C defined above. Let us plot $1 - \tilde{c}/c$ in polar coordin
$C_{\max}, 0.9C_{\max}, 0.5C_{\max}, 0.2C_{\max}$. The program below does the somewha

ork in Matplotlib, and the result appears in Figure 7. From the figure we early see that the maximum C value gives the best results, and that waves hose propagation direction makes an angle of 45 degrees with an axis are the ıost accurate.

```
def dispersion_relation_2D(kh, theta, C):
    arg = C*sqrt(sin(0.5*kh*cos(theta))**2 +
                 sin(0.5*kh*sin(theta))**2)
    c_frac = 2./(C*kh)*arcsin(arg)

    return c_frac

from numpy import exp, sin, cos, linspace, \
     pi, meshgrid, arcsin, sqrt
r = kh = linspace(0.001, pi, 101)
theta = linspace(0, 2*pi, 51)
r, theta = meshgrid(r, theta)

# Make 2x2 filled contour plots for 4 values of C
import matplotlib.pyplot as plt
C_max = 1/sqrt(2)
C = [[C_max, 0.9*C_max], [0.5*C_max, 0.2*C_max]]
fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
for row in range(2):
    for column in range(2):
        error = 1 - dispersion_relation_2D(
            kh, theta, C[row][column])
        print error.min(), error.max()
        cax = axes[row][column].contourf(
            theta, r, error, 50, vmin=0, vmax=0.36)
        axes[row][column].set_xticks([])
        axes[row][column].set_yticks([])

# Add colorbar to the last plot
cbar = plt.colorbar(cax)
cbar.ax.set_ylabel('error in wave velocity')
plt.savefig('disprel2D.png')
plt.savefig('disprel2D.pdf')
plt.show()
```

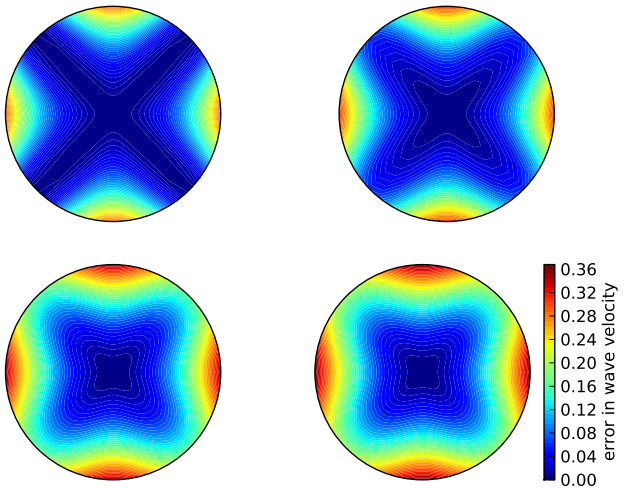



Figure 7: Error in numerical dispersion in 2D.

11 Finite difference methods for 2D and 3D equations

A natural next step is to consider extensions of the methods for varic ants of the one-dimensional wave equation to two-dimensional (2D) an dimensional (3D) versions of the wave equation.

11.1 Multi-dimensional wave equations

The general wave equation in d space dimensions, with constant wave ve can be written in the compact form

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0,T]\,.$$

In a 2D problem ($d = 2$),

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

while in three space dimensions ($d = 3$),

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\,.$$

Many applications involve variable coefficients, and the general wave equation
ı d dimensions is in this case written as

$$\varrho\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(q\nabla u) + f \text{ for } \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\ t\in(0,T], \tag{98}$$

hich in 2D becomes

$$\varrho(x,y)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t)\,. \tag{99}$$

o save some writing and space we may use the index notation, where subscript
x, y, or z means differentiation with respect to that coordinate. For example,

$$\frac{\partial^2 u}{\partial t^2} = u_{tt},$$
$$\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) = (qu_y)_y\,.$$

he 3D versions of the two model PDEs, with and without variable coefficients,
an with now with the aid of the index notation for differentiation be stated as

$$u_{tt} = c^2(u_{xx}+u_{yy}+u_{zz}) + f, \tag{100}$$
$$\varrho u_{tt} = (qu_x)_x + (qu_z)_z + (qu_z)_z + f\,. \tag{101}$$

At *each point* of the boundary $\partial\Omega$ of Ω we need *one* boundary condition
ıvolving the unknown u. The boundary conditions are of three principal types:

1. u is prescribed ($u=0$ or a known time variation for an incoming wave),
2. $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ prescribed (zero for reflecting boundaries),
3. an open boundary condition (also called radiation condition) is specified to let waves travel undisturbed out of the domain, see Exercise **??** for details.

ll the listed wave equations with *second-order* derivatives in time need *two*
ıitial conditions:

1. $u = I$,
2. $u_t = V$.

11.2 Mesh

We introduce a mesh in time and in space. The mesh in time consists
points

$$t_0 = 0 < t_1 < \cdots < t_{N_t},$$

often with a constant spacing $\Delta t = t_{n+1} - t_n$, $n\in\mathcal{I}_t^-$.

Finite difference methods are easy to implement on simple rectangle-
shaped domains. More complicated shapes of the domain require subst
more advanced techniques and implementational efforts. On a recta
box-shaped domain mesh points are introduced separately in the variou
directions:

$$x_0 < x_1 < \cdots < x_{N_x} \text{ in } x \text{ direction},$$
$$y_0 < y_1 < \cdots < y_{N_y} \text{ in } y \text{ direction},$$
$$z_0 < z_1 < \cdots < z_{N_z} \text{ in } z \text{ direction}\,.$$

We can write a general mesh point as (x_i, y_j, z_k, t_n), with $i\in\mathcal{I}_x$, $j\in\mathcal{I}_y$
and $n\in\mathcal{I}_t$.

It is a very common choice to use constant mesh spacings: $\Delta x = x_i$
$i\in\mathcal{I}_x^-$, $\Delta y = y_{j+1} - y_j$, $j\in\mathcal{I}_y^-$, and $\Delta z = z_{k+1} - z_k$, $k\in\mathcal{I}_z^-$. With equ
spacings one often introduces $h = \Delta x = \Delta y = \Delta z$.

The unknown u at mesh point (x_i, y_j, z_k, t_n) is denoted by $u^n_{i,j,k}$.
problems we just skip the z coordinate (by assuming no variation
direction: $\partial/\partial z = 0$) and write $u^n_{i,j}$.

11.3 Discretization

Two- and three-dimensional wave equations are easily discretized by ass
building blocks for discretization of 1D wave equations, because th
dimensional versions just contain terms of the same type that occurs in

Discretizing the PDEs. Equation (100) can be discretized as

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu + D_zD_zu) + f]^n_{i,j,k}\,.$$

A 2D version might be instructive to write out in detail:

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu) + f]^n_{i,j,k},$$

which becomes

$$\frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2} = c^2\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + c^2\frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j}}{\Delta y^2}$$

ssuming as usual that all values at the time levels n and $n-1$ are known, we an solve for the only unknown $u^{n+1}_{i,j}$. The result can be compactly written as

$$u^{n+1}_{i,j} = 2u^n_{i,j} + u^{n-1}_{i,j} + c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{103}$$

As in the 1D case, we need to develop a special formula for $u^1_{i,j}$ where we ombine the general scheme for $u^{n+1}_{i,j}$, when $n=0$, with the discretization of the itial condition:

$$[D_{2t}u = V]^0_{i,j} \quad \Rightarrow \quad u^{-1}_{i,j} = u^1_{i,j} - 2\Delta t V_{i,j}\,.$$

he result becomes, in compact form,

$$u^{n+1}_{i,j} = u^n_{i,j} - 2\Delta V_{i,j} + \frac{1}{2}c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{104}$$

The PDE (101) with variable coefficients is discretized term by term using ıe corresponding elements from the 1D case:

$$[\varrho D_tD_tu = (D_x\overline{q}^xD_xu + D_y\overline{q}^yD_yu + D_z\overline{q}^zD_zu) + f]^n_{i,j,k}\,. \tag{105}$$

/hen written out and solved for the unknown $u^{n+1}_{i,j,k}$, one gets the scheme

$$\begin{aligned}
u^{n+1}_{i,j,k} &= -u^{n-1}_{i,j,k} + 2u^n_{i,j,k} + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i+1,j,k})(u^n_{i+1,j,k}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i-1,j,k}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i-1,j,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i,j+1,k})(u^n_{i,j+1,k}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j-1,k}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i,j-1,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i,j,k+1})(u^n_{i,j,k+1}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j,k-1}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i,j,k-1})) + \\
&+ \qquad \Delta t^2 f^n_{i,j,k}\,.
\end{aligned}$$

Also here we need to develop a special formula for $u^1_{i,j,k}$ by combining the cheme for $n=0$ with the discrete initial condition, which is just a matter of ıserting $u^{-1}_{i,j,k} = u^1_{i,j,k} - 2\Delta t V_{i,j,k}$ in the scheme and solving for $u^1_{i,j,k}$.

Handling boundary conditions where is u known. The schemes listed bove are valid for the internal points in the mesh. After updating these, we eed to visit all the mesh points at the boundaries and set the prescribed u alue.

Discretizing the Neumann condition. The condition $\partial u/\partial n = 0$ plemented in 1D by discretizing it with a $D_{2x}u$ centered difference, and th eliminating the fictitious u point outside the mesh by using the general at the boundary point. Alternatively, one can introduce ghost cells and a ghost value to for use in the Neumann condition. Exactly the same ic reused in multi dimensions.

Consider $\partial u/\partial n = 0$ at a boundary $y=0$. The normal direction is $-y$ direction, so

$$\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y},$$

and we set

$$[-D_{2y}u = 0]^n_{i,0} \quad \Rightarrow \quad \frac{u^n_{i,1} - u^n_{i,-1}}{2\Delta y} = 0\,.$$

From this it follows that $u^n_{i,-1} = u^n_{i,1}$. The discretized PDE at the bc point $(i,0)$ reads

$$\frac{u^{n+1}_{i,0} - 2u^n_{i,0} + u^{n-1}_{i,0}}{\Delta t^2} = c^2\frac{u^n_{i+1,0} - 2u^n_{i,0} + u^n_{i-1,0}}{\Delta x^2} + c^2\frac{u^n_{i,1} - 2u^n_{i,0} + u^n_{i,-}}{\Delta y^2}$$

We can then just insert $u^1_{i,1}$ for $u^n_{i,-1}$ in this equation and then solve boundary value $u^{n+1}_{i,0}$ as done in 1D.

From these calculations, we see a pattern: the general scheme ap the boundary $j=0$ too if we just replace $j-1$ by $j+1$. Such a pa particularly useful for implementations. The details follow from the ex 1D case in Section 6.3.

The alternative approach to eliminating fictitious values outside the to have $u^n_{i,-1}$ available as a ghost value. The mesh is extended with or line (2D) or plane (3D) of ghost cells at a Neumann boundary. In the example it means that we need a line ghost cells below the y axis. Th values must be updated according to $u^{n+1}_{i,-1} = u^{n+1}_{i,1}$.

12 Implementation

We shall now describe in detail various Python implementations for sc standard 2D, linear wave equation with constant wave velocity and $u =$ (boundary. The wave equation is to be solved in the space-time domain Ω where $\Omega = (0,L_x)\times(0,L_y)$ is a rectangular spatial domain. More precis complete initial-boundary value problem is defined by

$$\begin{aligned}
u_t &= c^2(u_{xx}+u_{yy}) + f(x,y,t), && (x,y)\in\Omega,\ t\in(0,T],\\
u(x,y,0) &= I(x,y), && (x,y)\in\Omega,\\
u_t(x,y,0) &= V(x,y), && (x,y)\in\Omega,\\
u &= 0, && (x,y)\in\partial\Omega,\ t\in(0,T],
\end{aligned}$$

here $\partial\Omega$ is the boundary of Ω, in this case the four sides of the rectangle $), L_x] \times [0, L_y]$: $x = 0$, $x = L_x$, $y = 0$, and $y = L_y$.

The PDE is discretized as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]^n_{i,j},$$

hich leads to an explicit updating formula to be implemented in a program:

$$u^{n+1} = -u^{n-1}_{i,j} + 2u^n_{i,j} + C^2_x(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}) + C^2_y(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}) + \Delta t^2 f^n_{i,j}, \tag{110}$$

or all interior mesh points $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$, and for $n \in \mathcal{I}^+_t$. The constants $_x$ and C_y are defined as

$$C_x = c\frac{\Delta t}{\Delta x}, \quad C_x = c\frac{\Delta t}{\Delta y}.$$

At the boundary we simply set $u^{n+1}_{i,j} = 0$ for $i = 0$, $j = 0, \ldots, N_y$; $i = N_x$, $= 0, \ldots, N_y$; $j = 0$, $i = 0, \ldots, N_x$; and $j = N_y$, $i = 0, \ldots, N_x$. For the first ep, $n = 0$, (111) is combined with the discretization of the initial condition $_t = V$, $[D_{2t}u = V]^0_{i,j}$ to obtain a special formula for $u^1_{i,j}$ at the interior mesh oints:

$$u^1 = u^0_{i,j} + \Delta t V_{i,j} + \frac{1}{2}C^2_x(u^0_{i+1,j} - 2u^0_{i,j} + u^0_{i-1,j}) + \frac{1}{2}C^2_y(u^0_{i,j+1} - 2u^0_{i,j} + u^0_{i,j-1}) + \frac{1}{2}\Delta t^2 f^n_{i,j}, \tag{111}$$

The algorithm is very similar to the one in 1D:

1. Set initial condition $u^0_{i,j} = I(x_i, y_j)$
2. Compute $u^1_{i,j}$ from (111)
3. Set $u^1_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$
4. For $n = 1, 2, \ldots, N_t$:
   (a) Find $u^{n+1}_{i,j}$ from (111) for all internal mesh points, $i \in \mathcal{I}^i_x$, $j \in \mathcal{I}^i_y$
   (b) Set $u^{n+1}_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

2.1 Scalar computations

he `solver` function for a 2D case with constant wave velocity and $u = 0$ as oundary condition follows the setup from the similar function for the 1D case ı `wave1D_u0.py`, but there are a few necessary extensions. The code is in the rogram `wave2D_u0.py`[17].

[17] `http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`

Domain and mesh. The spatial domain is now $[0, L_x] \times [0, L_y]$, spec the arguments `Lx` and `Ly`. Similarly, the number of mesh points in the directions, N_x and N_y, become the arguments `Nx` and `Ny`. In multi-dim problems it makes less sense to specify a Courant number as the wave is a vector and the mesh spacings may differ in the various spatial di We therefore give Δt explicitly. The signature of the `solver` function i

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Key parameters used in the calculations are created as

```
x = linspace(0, Lx, Nx+1)                  # mesh points in x dir
y = linspace(0, Ly, Ny+1)                  # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)                 # mesh points in time
Cx2 = (c*dt/dx)**2;  Cy2 = (c*dt/dy)**2    # help variables
dt2 = dt**2
```

Solution arrays. We store $u^{n+1}_{i,j}$, $u^n_{i,j}$, and $u^{n-1}_{i,j}$ in three two-dim arrays,

```
u   = zeros((Nx+1,Ny+1))   # solution array
u_1 = zeros((Nx+1,Ny+1))   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1))   # solution at t-2*dt
```

where $u^{n+1}_{i,j}$ corresponds to `u[i,j]`, $u^n_{i,j}$ to `u_1[i,j]`, and $u^{n-1}_{i,j}$ to `u_2`

Index sets. It is also convenient to introduce the index sets (cf. Sect

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])
```

Computing the solution. Inserting the initial condition `I` in `u_1` and a callback to the user in terms of the `user_action` function is a straight generalization of the 1D code from Section 1.6:

```
for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, x, xv, y, yv, t, 0)
```

he `user_action` function has additional arguments compared to the 1D case. he arguments `xv` and `yv` fact will be commented upon in Section 12.2.

The key finite difference formula (103) for updating the solution at a time vel is implemented in a separate function as

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

he `step1` variable has been introduced to allow the formula to be reused for rst step $u^1_{i,j}$:

```
u = advance_scalar(u, u_1, u_2, f, x, y, t,
                   n, Cx2, Cy2, dt, V, step1=True)
```

elow, we will make many alternative implementations of the `advance_scalar` unction to speed up the code since most of the CPU time in simulations is spent n this function.

Finally, we remark that the `solver` function in the `wave2D_u0.py` code pdates arrays for the next time step by switching references as described in ection 4.5. If the solution `u` is return from `solver`, which it is not, it is important o set `u = u_1` after the time loop, otherwise `u` actually contains `u_2`.

2.2 Vectorized computations

he scalar code above turns out to be extremely slow for large 2D meshes, and robably useless in 3D beyond debugging of small test cases. Vectorization is herefore a must for multi-dimensional finite difference computations in Python. or example, with a mesh consisting of 30×30 cells, vectorization brings down he CPU time by a factor of 70 (!).

In the vectorized case we must be able to evaluate user-given functi $I(x, y)$ and $f(x, y, t)$, provided as Python functions `I(x,y)` and `f(x,y` the entire mesh in one array operation. Having the one-dimensional coo arrays `x` and `y` is not sufficient: these must be extended to vectorized v

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]
# or
xv = x.reshape((x.size, 1))
yv = y.reshape((1, y.size))
```

This is a standard required technique when evaluating functions over a 2 say `sin(xv)*cos(xv)`, which then gives a result with shape `(Nx+1,Ny+`

With the `xv` and `yv` arrays for vectorized computing, setting the condition is just a matter of

```
u_1[:,:] = I(xv, yv)
```

One could also have written `u_1 = I(xv, yv)` and let `u_1` point to a new but vectorized operations often makes use of direct insertion in the origin through `u_1[:,:]` because sometimes not all of the array is to be filled a function evaluation. This is the case with the computational scheme f

```
def advance_vectorized(u, u_1, u_2, f_a, Cx2, Cy2, dt2,
                       V=None, step1=False):
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    u_xx = u_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
    u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
    u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[1:-1,1:-1] + \
                   Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]
    if step1:
        u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
    # Boundary condition u=0
    j = 0
    u[:,j] = 0
    j = u.shape[1]-1
    u[:,j] = 0
    i = 0
    u[i,:] = 0
    i = u.shape[0]-1
    u[i,:] = 0
    return u
```

Array slices in 2D are more complicated to understand than those in the logic from 1D applies to each dimension separately. For example, whe $u^n_{i,j} - u^n_{i-1,j}$ for $i \in \mathcal{I}^+_x$, we just keep `j` constant and make a slice in index: `u_1[1:,j] - u_1[:-1,j]`, exactly as in 1D. The `1:` slice spec the indices $i = 1, 2, \ldots, N_x$ (up to the last valid index), while `:-1` spec

elevant indices for the second term: $0, 1, \ldots, N_x - 1$ (up to, but not including ne last index).

In the above code segment, the situation is slightly more complicated, because ach displaced slice in one direction is accompanied by a `1:-1` slice in the other irection. The reason is that we only work with the internal points for the index nat is kept constant in a difference.

The boundary conditions along the four sides makes use of a slice consisting f all indices along a boundary:

```
u[: ,0] = 0
u[:,Ny] = 0
u[0 ,:] = 0
u[Nx,:] = 0
```

The `f` function is in the above vectorized update of `u` first computed as an rray over all mesh points:

```
f_a = f(xv, yv, t[n])
```

Ve could, alternatively, used the call `f(xv, yv, t[n])[1:-1,1:-1]` in the last erm of the update statement, but other implementations in compiled languages enefit from having `f` available in an array rather than calling our Python unction `f(x,y,t)` for every point.

Also in the `advance_vectorized` function we have introduced a boolean `tep1` to reuse the formula for the first time step in the same way as we did ith `advance_scalar`. We refer to the `solver` function in `wave2D_u0.py` for ne details on how the overall algorithm is implemented.

The callback function now has the arguments `u, x, xv, y, yv, t, n`. he inclusion of `xv` and `yv` makes it easy to, e.g., compute an exact 2D so- ution in the callback function and compute errors, through an expression like `- u_exact(xv, yv, t[n])`.

2.3 Verification

esting a quadratic solution. The 1D solution from Section 2.4 can be eneralized to multi-dimensions and provides a test case where the exact solution lso fulfills the discrete equations such that we know (to machine precision) hat numbers the solver function should produce. In 2D we use the following eneralization of (30):

$$u_{\mathrm{e}}(x, y, t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t)\,. \tag{112}$$

his solution fulfills the PDE problem if $I(x, y) = u_{\mathrm{e}}(x, y, 0)$, $V = \frac{1}{2}u_{\mathrm{e}}(x, y, 0)$, nd $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$. To show that u_{e} also solves the iscrete equations, we start with the general results $[D_tD_t 1]^n = 0$, $[D_tD_t t]^n = 0$, nd $[D_tD_t t^2] = 2$, and use these to compute

$$D_xD_xu_{\mathrm{e}}]^n_{i,j} = [y(L_y - y)(1 + \frac{1}{2}t)D_xD_x x(L_x - x)]^n_{i,j} = y_j(L_y - y_j)(1 + \frac{1}{2}t_n)2\,.$$

A similar calculation must be carried out for the $[D_yD_yu_{\mathrm{e}}]^n_{i,j}$ and $[D_t$ terms. One must also show that the quadratic solution fits the special for $u^1_{i,j}$. The details are left as Exercise 11. The `test_quadratic` fun the `wave2D_u0.py`[18] program implements this verification as a nose tes

13 Migrating loops to Cython

Although vectorization can bring down the CPU time dramatically compa scalar code, there is still some factor 5-10 to win in these types of applica implementing the finite difference scheme in compiled code, typically in C, or C++. This can quite easily be done by adding a little extra cod program. Cython is an extension of Python that offers the easiest way our Python loops in the scalar code down to machine code and the effic C.

Cython can be viewed as an extended Python language where varia declared with types and where functions are marked to be implement Migrating Python code to Cython is done by copying the desired code s to functions (or classes) and placing them in one or more separate fi extension `.pyx`.

13.1 Declaring variables and annotating the code

Our starting point is the plain `advance_scalar` function for a scalar im tation of the updating algorithm for new values $u^{n+1}_{i,j}$:

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n]
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
```

[18] `http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`

```
    for j in Iy: u[i,j] = 0
    return u
```

We simply take a copy of this function and put it in a file `wave2D_u0_loop_cy.pyx`. he relevant Cython implementation arises from declaring variables with types nd adding some important annotations to speed up array computing in Cython. et us first list the complete code in the `.pyx` file:

```
import numpy as np
cimport numpy as np
cimport cython
ctypedef np.float64_t DT    # data type

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef advance(
    np.ndarray[DT, ndim=2, mode='c'] u,
    np.ndarray[DT, ndim=2, mode='c'] u_1,
    np.ndarray[DT, ndim=2, mode='c'] u_2,
    np.ndarray[DT, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):

    cdef:
        int Ix_start = 0
        int Iy_start = 0
        int Ix_end = u.shape[0]-1
        int Iy_end = u.shape[1]-1
        int i, j
        double u_xx, u_yy

    for i in range(Ix_start+1, Ix_end):
        for j in range(Iy_start+1, Iy_end):
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
    # Boundary condition u=0
    j = Iy_start
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    j = Iy_end
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    i = Ix_start
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    i = Ix_end
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    return u
```

This example may act as a recipe on how to transform array-intensive code ith loops into Cython.

1. Variables are declared with types: for example, `double v` in the argument list instead of just `v`, and `cdef double v` for a variable `v` in the body of the function. A Python `float` object is declared as `double` for translation to C by Cython, while an `int` object is declared by `int`.

2. Arrays need a comprehensive type declaration involving

   - the type `np.ndarray`,
   - the data type of the elements, here 64-bit floats, abbreviate through `ctypedef np.float64_t DT` (instead of `DT` we could full name of the data type: `np.float64_t`, which is a Cython type),
   - the dimensions of the array, here `ndim=2` and `ndim=1`,
   - specification of contiguous memory for the array (`mode='c'`)

3. Functions declared with `cpdef` are translated to C but also accessil Python.

4. In addition to the standard `numpy` import we also need a special import of `numpy`: `cimport numpy as np`, to appear *after* the s import.

5. By default, array indices are checked to be within their legal lin speed up the code one should turn off this feature for a specific f by placing `@cython.boundscheck(False)` above the function hea

6. Also by default, array indices can be negative (counting from the e this feature has a performance penalty and is therefore here turne writing `@cython.wraparound(False)` right above the function he

7. The use of index sets `Ix` and `Iy` in the scalar code cannot be fully translated to C. One reason is that constructions like `Ix` involve negative indices, and these are now turned off. Another is that Cython loops must take the form `for i in xrange` or `fo range` for being translated into efficient C loops. We have there troduced `Ix_start` as `Ix[0]` and `Ix_end` as `Ix[-1]` to hold th and end of the values of index i. Similar variables are introdu the j index. A loop `for i in Ix` is with these new variables wr `for i in range(Ix_start, Ix_end+1)`.

Array declaration syntax in Cython.

We have used the syntax `np.ndarray[DT, ndim=2, mode='c']` to de `numpy` arrays in Cython. There is a simpler, alternative syntax, empl typed memory views[a], where the declaration looks like `double [` However, the full support for this functionality is not yet ready, and i text we use the full array declaration syntax.

[a] `http://docs.cython.org/src/userguide/memoryviews.html`

3.2 Visual inspection of the C translation

ython can visually explain how successfully it can translate a code from Python
C. The command

```
erminal> cython -a wave2D_u0_loop_cy.pyx
```

roduces an HTML file `wave2D_u0_loop_cy.html`, which can be loaded into a eb browser to illustrate which lines of the code that have been translated to C. igure 8 shows the illustrated code. Yellow lines indicate the lines that Cython id not manage to translate to efficient C code and that remain in Python. For ıe present code we see that Cython is able to translate all the loops with array ɔmputing to C, which is our primary goal.

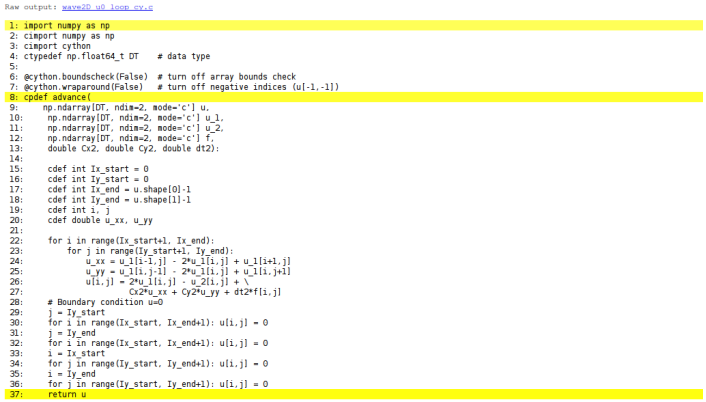


Figure 8: Visual illustration of Cython's ability to translate Python to C.

You can also inspect the generated C code directly, as it appears in the file `ave2D_u0_loop_cy.c`. Nevertheless, understanding this C code requires some ımiliarity with writing Python extension modules in C by hand. Deep down in ıe file we can see in detail how the compute-intensive statements are translated ɔme complex C code that is quite different from what we a human would write ıt least if a direct correspondence to the mathematics was in mind).

3.3 Building the extension module

ython code must be translated to C, compiled, and linked to form what is known ı the Python world as a *C extension module*. This is usually done by making a `etup.py` script, which is the standard way of building and installing Python ɔftware. For an extension module arising from Cython code, the following `etup.py` script is all we need to build and install the module:


```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

cymodule = 'wave2D_u0_loop_cy'
setup(
  name=cymodule
  ext_modules=[Extension(cymodule, [cymodule + '.pyx'],)],
  cmdclass={'build_ext': build_ext},
)
```

We run the script by

```
Terminal> python setup.py build_ext --inplace
```

The `-inplace` option makes the extension module available in the directory as the file `wave2D_u0_loop_cy.so`. This file acts as a normal module that can be imported and inspected:

```
>>> import wave2D_u0_loop_cy
>>> dir(wave2D_u0_loop_cy)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__test__', 'advance', 'np']
```

The important output from the `dir` function is our Cython function a (the module also features the imported `numpy` module under the name `np` as many standard Python objects with double underscores in their nam

The `setup.py` file makes use of the `distutils` package in Pyth Cython's extension of this package. These tools know how Python was the computer and will use compatible compiler(s) and options when l other code in Cython, C, or C++. Quite some experience with buildi program systems is needed to do the build process manually, so using a `se` script is strongly recommended.

Simplified build of a Cython module.

When there is no need to link the C code with special libraries, Cy offers a shortcut for generating and importing the extension module:

```
import pyximport; pyximport.install()
```

This makes the `setup.py` script redundant. However, in the `wave2D_u` code we do not use `pyximport` and require an explicit build process o and many other modules.

3.4 Calling the Cython function from Python

he `wave2D_u0_loop_cy` module contains our `advance` function, which we now ıay call from the Python program for the wave equation:

```
ımport wave2D_u0_loop_cy
ıdvance = wave2D_u0_loop_cy.advance
...
for n in It[1:-1:                    # time loop
    f_a[:,:] = f(xv, yv, t[n])       # precompute, size as u
    u = advance(u, u_1, u_2, f_a, x, y, t, Cx2, Cy2, dt2)
```

fficiency. For a mesh consisting of 120×120 cells, the scalar Python code equire 1370 CPU time units, the vectorized version requires 5.5, while the ython version requires only 1! For a smaller mesh with 60×60 cells Cython is bout 1000 times faster than the scalar Python code, and the vectorized version about 6 times slower than the Cython version.

.4 Migrating loops to Fortran

ıstead of relying on Cython's (excellent) ability to translate Python to C, we an invoke a compiled language directly and write the loops ourselves. Let us tart with Fortran 77, because this is a language with more convenient array andling than C (or plain C++). Or more precisely, we can with ease program ith the same multi-dimensional indices in the Fortran code as in the `numpy` rrays in the Python code, while in C these arrays are one-dimensional and equires us to reduce multi-dimensional indices to a single index.

4.1 The Fortran subroutine

Ve write a Fortran subroutine `advance` in a file `wave2D_u0_loop_f77.f`[19] for nplementing the updating formula (111) and setting the solution to zero at the oundaries:

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
      integer Nx, Ny
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx,0:Ny), Cx2, Cy2, dt2
      integer i, j
      real*8 u_xx, u_yy
Cf2py intent(in, out) u

C     Scheme at interior points
      do j = 1, Ny-1
         do i = 1, Nx-1
            u_xx = u_1(i-1,j) - 2*u_1(i,j) + u_1(i+1,j)
            u_yy = u_1(i,j-1) - 2*u_1(i,j) + u_1(i,j+1)
            u(i,j) = 2*u_1(i,j) - u_2(i,j) + Cx2*u_xx + Cy2*u_yy +
     &               dt2*f(i,j)
         end do
      end do

C     Boundary conditions
      j = 0
      do i = 0, Nx
         u(i,j) = 0
      end do
      j = Ny
      do i = 0, Nx
         u(i,j) = 0
      end do
      i = 0
      do j = 0, Ny
         u(i,j) = 0
      end do
      i = Nx
      do j = 0, Ny
         u(i,j) = 0
      end do
      return
      end
```

This code is plain Fortran 77, except for the special `Cf2py` comment lin here specifies that `u` is both an input argument *and* an object to be r from the `advance` routine. Or more precisely, Fortran is not able return a from a function, but we need a *wrapper code* in C for the Fortran subro enable calling it from Python, and in this wrapper code one can return calling Python code.

Remark.

It is not strictly necessary to return `u` to the calling Python code the `advance` function will modify the elements of `u`, but the conventi Python is to get all output from a function as returned values. Th the right way of calling the above Fortran subroutine from Python is

```
u = advance(u, u_1, u_2, f, Cx2, Cy2, dt2)
```

The less encouraged style, which works and resembles the way the Fo subroutine is called from Fortran, reads

```
advance(u, u_1, u_2, f, Cx2, Cy2, dt2)
```

[19] `http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0_loop_f77.f`

4.2 Building the Fortran module with f2py

he nice feature of writing loops in Fortran is that the tool `f2py` can with very ttle work produce a C extension module such that we can call the Fortran ersion of `advance` from Python. The necessary commands to run are

```
erminal> f2py -m wave2D_u0_loop_f77 -h wave2D_u0_loop_f77.pyf \
         --overwrite-signature wave2D_u0_loop_f77.f
erminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
         -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

he first command asks `f2py` to interpret the Fortran code and make a Fortran 90 pecification of the extension module in the file `wave2D_u0_loop_f77.pyf`. The econd command makes `f2py` generate all necessary wrapper code, compile our ortran file and the wrapper code, and finally build the module. The build process akes place in the specified subdirectory `build_f77` so that files can be inspected something goes wrong. The option `-DF2PY_REPORT_ON_ARRAY_COPY=1` makes 2py write a message for every array that is copied in the communication between ortran and Python, which is very useful for avoiding unnecessary array copying ee below). The name of the module file is `wave2D_u0_loop_f77.so`, and this le can be imported and inspected as any other Python module:

```
>>> import wave2D_u0_loop_f77
>>> dir(wave2D_u0_loop_f77)
['__doc__', '__file__', '__name__', '__package__',
 '__version__', 'advance']
>>> print wave2D_u0_loop_f77.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py....
Functions:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,
      nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

Examine the doc strings!

Printing the doc strings of the module and its functions is extremely important after having created a module with `f2py`, because `f2py` makes Python interfaces to the Fortran functions that are different from how the functions are declared in the Fortran code (!). The rationale for this behavior is that `f2py` creates *Pythonic* interfaces such that Fortran routines can be called in the same way as one calls Python functions. Output data from Python functions is always returned to the calling code, but this is technically impossible in Fortran. Also, arrays in Python are passed to Python functions without their dimensions because that information is packed with the array data in the array objects, but this is not possible in Fortran. Therefore, `f2py` removes array dimensions from the argument list, and `f2py` makes it possible to return objects back to Python.

Let us follow the advice of examining the doc strings and take a cl at the documentation `f2py` has generated for our Fortran `advance` sub

```
>>> print wave2D_u0_loop_f77.advance.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py
Functions:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,
              nx=(shape(u,0)-1),ny=(shape(u,1)-1))
.
advance - Function signature:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,[nx,ny])
Required arguments:
  u : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_1 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_2 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  f : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  cx2 : input float
  cy2 : input float
  dt2 : input float
Optional arguments:
  nx := (shape(u,0)-1) input int
  ny := (shape(u,1)-1) input int
Return objects:
  u : rank-2 array('d') with bounds (nx + 1,ny + 1)
```

Here we see that the `nx` and `ny` parameters declared in Fortran are arguments that can be omitted when calling `advance` from Python.

We strongly recommend to print out the documentation of *every* function to be called from Python and make sure the call syntax is ex listed in the documentation.

14.3 How to avoid array copying

Multi-dimensional arrays are stored as a stream of numbers in memc a two-dimensional array consisting of rows and columns there are tv of creating such a stream: *row-major ordering*, which means that r stored consecutively in memory, or *column-major ordering*, which means columns are stored one after each other. All programming languages i from C, including Python, apply the row-major ordering, but Fortr column-major storage. Thinking of a two-dimensional array in Python o matrix, it means that Fortran works with the transposed matrix.

Fortunately, `f2py` creates extra code so that accessing `u(i,j)` in the subroutine corresponds to the element `u[i,j]` in the underlying `nump` (without the extra code, `u(i,j)` in Fortran would access `u[j,i]` in th array). Technically, `f2py` takes a copy of our `numpy` array and reorders t before sending the array to Fortran. Such copying can be costly. For 2 simulations on a 60×60 grid the overhead of copying is a factor of 5 means that almost the whole performance gain of Fortran over vectorize code is lost!

To avoid having `f2py` to copy arrays with C storage to the corres Fortran storage, we declare the arrays with Fortran storage:

```
order = 'Fortran' if version == 'f77' else 'C'
u   = zeros((Nx+1,Ny+1), order=order)   # solution array
u_1 = zeros((Nx+1,Ny+1), order=order)   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1), order=order)   # solution at t-2*dt
```

In the compile and build step of using `f2py`, it is recommended to add an extra option for making `f2py` report on array copying:

```
terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

It can sometimes be a challenge to track down which array that causes a copying. There are two principal reasons for copying array data: either the array does not have Fortran storage or the element types do not match those declared in the Fortran code. The latter cause is usually effectively eliminated by using `real*8` data in the Fortran code and `float64` (the default `float` type in `numpy`) in the arrays on the Python side. The former reason is more common, and to check whether an array before a Fortran call has the right storage one can print the result of `isfortran(a)`, which is `True` if the array `a` has Fortran storage.

Let us look at an example where we face problems with array storage. A typical problem in the `wave2D_u0.py` code is to set

```
f_a = f(xv, yv, t[n])
```

before the call to the Fortran `advance` routine. This computation creates a new array with C storage. An undesired copy of `f_a` will be produced when sending `f_a` to a Fortran routine. There are two remedies, either direct insertion of data in an array with Fortran storage,

```
f_a = zeros((Nx+1, Ny+1), order='Fortran')
...
f_a[:,:] = f(xv, yv, t[n])
```

or remaking the `f(xv, yv, t[n])` array,

```
f_a = asarray(f(xv, yv, t[n]), order='Fortran')
```

The former remedy is most efficient if the `asarray` operation is to be performed a large number of times.

Efficiency. The efficiency of this Fortran code is very similar to the Cython code. There is usually nothing more to gain, from a computational efficiency point of view, by implementing the *complete* Python program in Fortran or C. That will just be a lot more code for all administering work that is needed in scientific software, especially if we extend our sample program `wave2D_u0.py` to handle a real scientific problem. Then only a small portion will consist of loops

with intensive array calculations. These can be migrated to Cython or as explained, while the rest of the programming can be more convenient in Python.

15 Migrating loops to C via Cython

The computationally intensive loops can alternatively be implemented code. Just as Fortran calls for care regarding the storage of two-dime arrays, working with two-dimensional arrays in C is a bit tricky. The r that `numpy` arrays are viewed as one-dimensional arrays when transferr while C programmers will think of `u`, `u_1`, and `u_2` as two dimensiona and index them like `u[i][j]`. The C code must declare `u` as `double*` translate an index pair `[i][j]` to a corresponding single index when `u` is as one-dimensional. This translation requires knowledge of how the nun `u` are stored in memory.

15.1 Translating index pairs to single indices

Two-dimensional `numpy` arrays with the default C storage are stored row In general, multi-dimensional arrays with C storage are stored such that index has the fastest variation, then the next last index, and so on, en with the slowest variation in the first index. For a two-dimensional `u` c as `zeros((Nx+1,Ny+1))` in Python, the individual elements are stored following order:

```
u[0,0], u[0,1], u[0,2], ..., u[0,Ny], u[1,0], u[1,1], ...,
u[1,Ny], u[2,0], ..., u[Nx,0], u[Nx,1], ..., u[Nx, Ny]
```

Viewing `u` as one-dimensional, the index pair (i, j) translates to $i(N_y$ So, where a C programmer would naturally write an index `u[i][j]`, the i must read `u[i*(Ny+1) + j]`. This is tedious to write, so it can be ha define a C macro,

```
#define idx(i,j) (i)*(Ny+1) + j
```

so that we can write `u[idx(i,j)]`, which reads much better and is e debug.

Be careful with macro definitions.

Macros just perform simple text substitutions: `idx(hello,world)` panded to `(hello)*(Ny+1) + world`. The parenthesis in `(i)` are esse - using the natural mathematical formula `i*(Ny+1) + j` in the m definition, `idx(i-1,j)` would expand to `i-1*(Ny+1) + j`, which i

wrong formula. Macros are handy, but requires careful use. In C++, inline functions are safer and replace the need for macros.

5.2 The complete C code

he C version of our function `advance` can be coded as follows.

```
#define idx(i,j) (i)*(Ny+1) + j

void advance(double* u, double* u_1, double* u_2, double* f,
             double Cx2, double Cy2, double dt2, int Nx, int Ny)
{
  int i, j;
  double u_xx, u_yy;
  /* Scheme at interior points */
  for (i=1; i<=Nx-1; i++) {
    for (j=1; j<=Ny-1; j++) {
      u_xx = u_1[idx(i-1,j)] - 2*u_1[idx(i,j)] + u_1[idx(i+1,j)];
      u_yy = u_1[idx(i,j-1)] - 2*u_1[idx(i,j)] + u_1[idx(i,j+1)];
      u[idx(i,j)] = 2*u_1[idx(i,j)] - u_2[idx(i,j)] +
        Cx2*u_xx + Cy2*u_yy + dt2*f[idx(i,j)];
    }
  }
  /* Boundary conditions */
  j = 0;  for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  j = Ny; for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  i = 0;  for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
  i = Nx; for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
}
```

5.3 The Cython interface file

ll the code above appears in a file `wave2D_u0_loop_c.c`[20]. We need to compile nis file together with C wrapper code such that `advance` can be called from ython. Cython can be used to generate appropriate wrapper code. The relevant ython code for interfacing C is placed in a file with extension `.pyx`. Here this le, called `wave2D_u0_loop_c_cy.pyx`[21], looks like

```
import numpy as np
cimport numpy as np
cimport cython

cdef extern from "wave2D_u0_loop_c.h":
    void advance(double* u, double* u_1, double* u_2, double* f,
                 double Cx2, double Cy2, double dt2,
                 int Nx, int Ny)

@cython.boundscheck(False)
@cython.wraparound(False)
def advance_cwrap(
```

[20] http://tinyurl.com/nm5587k/wave//wave2D_u0/wave2D_u0_loop_c.c
[21] http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0_loop_c_cy.pyx


```
    np.ndarray[double, ndim=2, mode='c'] u,
    np.ndarray[double, ndim=2, mode='c'] u_1,
    np.ndarray[double, ndim=2, mode='c'] u_2,
    np.ndarray[double, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
    advance(&u[0,0], &u_1[0,0], &u_2[0,0], &f[0,0],
            Cx2, Cy2, dt2,
            u.shape[0]-1, u.shape[1]-1)
    return u
```

We first declare the C functions to be interfaced. These must also appea header file, `wave2D_u0_loop_c.h`[22],

```
extern void advance(double* u, double* u_1, double* u_2, double*
                    double Cx2, double Cy2, double dt2,
                    int Nx, int Ny);
```

The next step is to write a Cython function with Python objects as arg The name `advance` is already used for the C function so the function to b from Python is named `advance_cwrap`. The contents of this function is a call to the `advance` version in C. To this end, the right information f Python objects must be passed on as arguments to `advance`. Arrays with their C pointers to the first element, obtained in Cython as `&u[0,` `&` takes the address of a C variable). The `Nx` and `Ny` arguments in `adva` easily obtained from the shape of the `numpy` array `u`. Finally, `u` must be r such that we can set `u = advance(...)` in Python.

15.4 Building the extension module

It remains to build the extension module. An appropriate `setup.py` fil

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

sources = ['wave2D_u0_loop_c.c', 'wave2D_u0_loop_c_cy.pyx']
module = 'wave2D_u0_loop_c_cy'
setup(
  name=module,
  ext_modules=[Extension(module, sources,
                         libraries=[], # C libs to link with
                         )],
  cmdclass={'build_ext': build_ext},
)
```

All we need to specify is the `.c` file(s) and the `.pyx` interface file. Cyth tomatically run to generate the necessary wrapper code. Files are then c and linked to an extension module residing in the file `wave2D_u0_loop_c` Here is a session with running `setup.py` and examining the resulting m Python

[22] http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0_loop_c.h

```
erminal> python setup.py build_ext --inplace
erminal> python
>> import wave2D_u0_loop_c_cy as m
>> dir(m)
'__builtins__', '__doc__', '__file__', '__name__', '__package__',
'__test__', 'advance_cwrap', 'np']
```

he call to the C version of `advance` can go like this in Python:

```
import wave2D_u0_loop_c_cy
advance = wave2D_u0_loop_c_cy.advance_cwrap
...
f_a[:,:] = f(xv, yv, t[n])
u = advance(u, u_1, u_2, f_a, Cx2, Cy2, dt2)
```

fficiency. In this example, the C and Fortran code runs at the same speed, nd there are no significant differences in the efficiency of the wrapper code. The verhead implied by the wrapper code is negligible as long as we do not work ith very small meshes and consequently little numerical work in the `advance` nction.

.6 Migrating loops to C via f2py

n alternative to using Cython for interfacing C code is to apply `f2py`. The C ode is the same, just the details of specifying how it is to be called from Python iffer. The `f2py` tool requires the call specification to be a Fortran 90 module efined in a `.pyf` file. This file was automatically generated when we interfaced Fortran subroutine. With a C function we need to write this module ourselves, r we can use a trick and let `f2py` generate it for us. The trick consists in writing ne signature of the C function with Fortran syntax and place it in a Fortran le, here `wave2D_u0_loop_c_f2py_signature.f`:

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
Cf2py intent(c) advance
      integer Nx, Ny, N
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
Cf2py intent(in, out) u
Cf2py intent(c) u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny
      return
      end
```

ote that we need a special `f2py` instruction, through a `Cf2py` comment line, for elling that all the function arguments are C variables. We also need to specify nat the function is actually in C: `intent(c) advance`.

Since `f2py` is just concerned with the function signature and not the complete ontents of the function body, it can easily generate the Fortran 90 module pecification based solely on the signature above:


```
Terminal> f2py -m wave2D_u0_loop_c_f2py \
          -h wave2D_u0_loop_c_f2py.pyf --overwrite-signature \
          wave2D_u0_loop_c_f2py_signature.f
```

The compile and build step is as for the Fortran code, except that we lis instead of Fortran files:

```
Terminal> f2py -c wave2D_u0_loop_c_f2py.pyf \
          --build-dir tmp_build_c \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_c.c
```

As when interfacing Fortran code with `f2py`, we need to print out the do to see the exact call syntax from the Python side. This doc string is i for the C and Fortran versions of `advance`.

16.1 Migrating loops to C++ via f2py

C++ is a much more versatile language than C or Fortran and has over two decades become very popular for numerical computing. Many will t prefer to migrate compute-intensive Python code to C++. This is, in p easy: just write the desired C++ code and use some tool for interf from Python. A tool like SWIG[23] can interpret the C++ code and g interfaces for a wide range of languages, including Python, Perl, Ruby, a However, SWIG is a comprehensive tool with a correspondingly steep l curve. Alternative tools, such as Boost Python[24], SIP[25], and Shibok similarly comprehensive. Simpler tools include PyBindGen[27],

A technically much easier way of interfacing C++ code is to d possibility to use C++ classes directly from Python, but instead ma interface to the C++ code. The C interface can be handled by `f2py` a in the example with pure C code. Such a solution means that classes in and C++ cannot be mixed and that only primitive data types like n strings, and arrays can be transferred between Python and C++. Actua is often a very good solution because it forces the C++ code to work c data, which usually gives faster code than if fancy data structures with are used. The arrays coming from Python, and looking like plain C/C++ can be efficiently wrapped in more user-friendly C++ array classes in t code, if desired.

[23] http://swig.org/
[24] http://www.boost.org/doc/libs/1_51_0/libs/python/doc/index.html
[25] http://riverbankcomputing.co.uk/software/sip/intro
[26] http://qt-project.org/wiki/Category:LanguageBindings::PySide::Shiboker
[27] http://code.google.com/p/pybindgen/

.7 Using classes to implement a simulator

- Introduce classes `Mesh`, `Function`, `Problem`, `Solver`, `Visualizer`, `File`

.8 Exercises

Exercise 11: Check that a solution fulfills the discrete model

Carry out all mathematical details to show that (112) is indeed a solution of the discrete model for a 2D wave equation with $u = 0$ on the boundary. One must check the boundary conditions, the initial conditions, the general discrete equation at a time level and the special version of this equation for the first time level. Filename: `check_quadratic_solution.pdf`.

Project 12: Calculus with 2D/3D mesh functions

The goal of this project is to redo Project 5 with 2D and 3D mesh functions ($f_{i,j}$ and $f_{i,j,k}$).

Differentiation. The differentiation results in a discrete gradient function, which in the 2D case can be represented by a three-dimensional array `df[d,i,j]` where `d` represents the direction of the derivative and `i` and `j` are mesh point counters in 2D (the 3D counterpart is `df[d,i,j,k]`).

Integration. The integral of a 2D mesh function $f_{i,j}$ is defined as

$$F_{i,j} = \int_{y_0}^{y_j}\int_{x_0}^{x_i} f(x,y)dxdy,$$

where $f(x,y)$ is a function that takes on the values of the discrete mesh function $f_{i,j}$ at the mesh points, but can also be evaluated in between the mesh points. The particular variation between mesh points can be taken as bilinear, but this is not important as we will use a product Trapezoidal rule to approximate the integral over a cell in the mesh and then we only need to evaluate $f(x,y)$ at the mesh points.

Suppose $F_{i,j}$ is computed. The calculation of $F_{i+1,j}$ is then

$$\begin{aligned} F_{i+1,j} &= F_{i,j} + \int_{x_i}^{x_{i+1}}\int_{y_0}^{y_j} f(x,y)dydx \\ &\approx \Delta x \int_{y_0}^{y_j} f(x_{i+\frac{1}{2}},y)dy \\ &\approx \Delta x \frac{1}{2}\left(\int_{y_0}^{y_j} f(x_i,y)dy + \int_{y_0}^{y_j} f(x_{i+1},y)dy\right) \end{aligned}$$

The integrals in the y direction can be approximated by a Trapezoidal similar idea can be used to compute $F_{i,j+1}$. Thereafter, $F_{i+1,j+1}$ can be co by adding the integral over the final corner cell to $F_{i+1,j} + F_{i,j+1} - F_{i,j}$ out the details of these computations and extend the ideas to 3D. Fi `mesh_calculus_3D.py`.

Exercise 13: Implement Neumann conditions in 2D

Modify the `wave2D_u0.py`[28] program, which solves the 2D wave equatic $c^2(u_{xx} + u_{yy})$ with constant wave velocity c and $u = 0$ on the boun have Neumann boundary conditions: $\partial u/\partial n = 0$. Include both scalar c debugging and reference) and vectorized code (for speed).

To test the code, use $u = 1.2$ as solution ($I(x,y) = 1.2$, $V = f = 0$ arbitrary), which should be exactly reproduced with any mesh as long stability criterion is satisfied. Another test is to use the plug-shaped puls `pulse` function from Section 8 and the `wave1D_dn_vc.py`[29] program. Th is exactly propagated in 1D if $c\Delta t/\Delta x = 1$. Check that also the 2D p can propagate this pulse exactly in x direction ($c\Delta t/\Delta x = 1$, Δy arbitra y direction ($c\Delta t/\Delta y = 1$, Δx arbitrary). Filename: `wave2D_dn.py`.

Exercise 14: Test the efficiency of compiled loops in

Extend the `wave2D_u0.py` code and the Cython, Fortran, and C version Set up an efficiency experiment to determine the relative efficiency of pu Python code, vectorized code, Cython-compiled loops, Fortran-compile and C-compiled loops. Normalize the CPU time for each mesh by the version. Filename: `wave3D_u0.py`.

19 Applications of wave equations

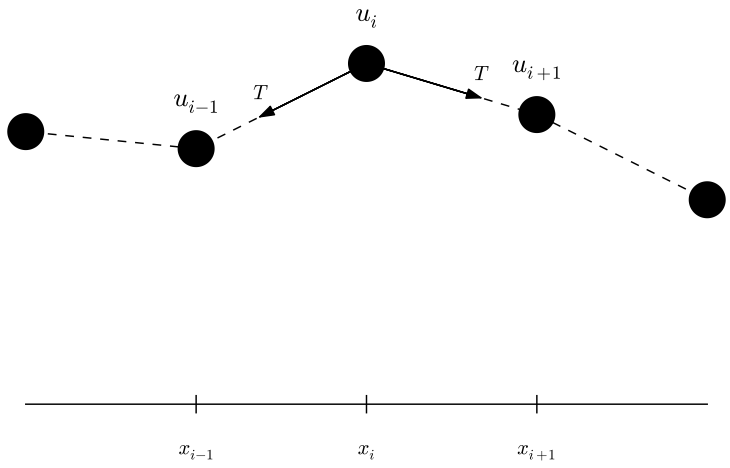
This section presents a range of wave equation models for different p phenomena. Although many wave motion problems in physics can be mo the standard linear wave equation, or a similar formulation with a system order equations, there are some exceptions. Perhaps the most important waves: these are modeled by the Laplace equation with time-dependent b conditions at the water surface (long water waves, however, can be appro by a standard wave equation, see Section 19.7). Quantum mechanica constitute another example where the waves are governed by the Schr equation and not a standard wave equation. Many wave phenomena al to take nonlinear effects into account when the wave amplitude is sig Shock waves in the air is a primary example.

The derivations in the following are very brief. Those with a firm bac in continuum mechanics will probably have enough information to fil

[28] `http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`
[29] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

etails, while other readers will hopefully get some impression of the physics and pproximations involved when establishing wave equation models.

9.1 Waves on a string



igure 9: Discrete string model with point masses connected by elastic strings.

Figure 9 shows a model we may use to derive the equation for waves on a ring. The string is modeled as a set of discrete point masses (at mesh points) ith elastic strings in between. The strings are at a high constant tension T. /e let the mass at mesh point x_i be m_i. The displacement of this mass point in direction is denoted by $u_i(t)$.

The motion of mass m_i is governed by Newton's second law of motion. The osition of the mass at time t is $x_i\boldsymbol{i} + u_i(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in ıe x and y direction, respectively. The acceleration is then $u_i''(t)\boldsymbol{j}$. Two forces

are acting on the mass as indicated in Figure 9. The force $\boldsymbol{T}^-$ acting tov point x_{i-1} can be decomposed as

$$\boldsymbol{T}^- = -T\sin\phi\boldsymbol{i} - T\cos\phi\boldsymbol{j},$$

where ϕ is the angle between the force and the line $x = x_i$. Let $\Delta u_i = u$ and let $\Delta s_i = \sqrt{\Delta u_i^2 + (x_i - x_{i-1})^2}$ be the distance from mass m_{i-1} m_i. It is seen that $\cos\phi = \Delta u_i/\Delta s_i$ and $\sin\phi = (x_i - x_{i-1})/\Delta s$ or Δs we introduce a constant mesh spacing $\Delta x = x_i - x_{i-1}$. The force can written

$$\boldsymbol{T}^- = -T\frac{\Delta x}{\Delta s_i}\boldsymbol{i} - T\frac{\Delta u_i}{\Delta s_i}\boldsymbol{j}\,.$$

The force $\boldsymbol{T}^+$ acting toward x_{i+1} can be calculated in a similar way:

$$\boldsymbol{T}^+ = T\frac{\Delta x}{\Delta s_{i+1}}\boldsymbol{i} + T\frac{\Delta u_{i+1}}{\Delta s_{i+1}}\boldsymbol{j}\,.$$

Newton's second law becomes

$$m_i u_i''(t)\boldsymbol{j} = \boldsymbol{T}^+ + \boldsymbol{T}^-,$$

which gives the component equations

$$T\frac{\Delta x}{\Delta s_i} = T\frac{\Delta x}{\Delta s_{i+1}},$$
$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta s_{i+1}} - T\frac{\Delta u_i}{\Delta s_i}\,.$$

A basic reasonable assumption for a string is small displacements small displacement gradients $\Delta u_i/\Delta x$. For small $g = \Delta u_i/\Delta x$ we have

$$\Delta s_i = \sqrt{\Delta u_i^2 + \Delta x^2} = \Delta x\sqrt{1+g^2} + \Delta x(1 + \frac{1}{2}g^2 + \mathcal{O}(g^4) \approx \Delta x$$

Equation (113) is then simply the identity $T = T$, while (114) can be wı

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta x} - T\frac{\Delta u_i}{\Delta x},$$

which upon division by Δx and introducing the density $\varrho_i = m_i/\Delta x$ be

$$\varrho_i u_i''(t) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,.$$

We can now choose to approximate u_i'' by a finite difference in time and discretized wave equation,

$$\varrho_i\frac{1}{\Delta t^2}\left(u_i^{n+1} - 2u_i^n - u_i^{n-1}\right) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,.$$

n the other hand, we may go to the continuum limit $\Delta x \to 0$ and replace $u_i(t)$ y $u(x,t)$, ϱ_i by $\varrho(x)$, and recognize that the right-hand side of (115) approaches $^2u/\partial x^2$ as $\Delta x \to 0$. We end up with the continuous model for waves on a ring:

$$\varrho\frac{\partial^2 u}{\partial t^2} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{117}$$

ote that the density ϱ may change along the string, while the tension T is a onstant. With variable wave velocity $c(x) = \sqrt{T/\varrho(x)}$ we can write the wave quation in the more standard form

$$\frac{\partial^2 u}{\partial t^2} = c^2(x)\frac{\partial^2 u}{\partial x^2}\,. \tag{118}$$

ecause of the way ϱ enters the equations, the variable wave velocity does *not* ppear inside the derivatives as in many other versions of the wave equation. owever, most strings of interest have constant ϱ.

The end point of a string are fixed so that the displacement u is zero. The oundary conditions are therefore $u = 0$.

amping. Air resistance and non-elastic effects in the string will contribute o reduce the amplitudes of the waves so that the motion dies out after some me. This damping effect can be modeled by a term bu_t on the left-hand side of ne equation

$$\varrho\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{119}$$

he parameter b must normally be determined from physical experiments.

xternal forcing. It is easy to include an external force acting on the string. ay we have a vertical force $\tilde{f}_i\boldsymbol{j}$ acting on mass m_i. This force affects the ertical component of Newton's law and gives rise to an extra term $\tilde{f}(x,t)$ n the right-hand side of (117). In the model (118) we would add a term $(x,t) = \tilde{f}(x,y)/\varrho(x)$.

Modeling the tension via springs. We assumed, in the derivation above, nat the tension in the string, T, was constant. It is easy to check this assumption y modeling the string segments between the masses as standard springs, where ne force (tension T) is proportional to the elongation of the spring segment. et k be the spring constant, and set $T_i = k\Delta\ell$ for the tension in the spring egment between x_{i-1} and x_i, where $\Delta\ell$ is the elongation of this segment from ne tension-free state. A basic feature of a string is that it has high tension in ne equilibrium position $u = 0$. Let the string segment have an elongation $\Delta\ell_0$ n the equilibrium position. After deformation of the string, the elongation is $\ell = \Delta\ell_0 + \Delta s_i$: $T_i = k(\Delta\ell_0 + \Delta s_i) \approx k(\Delta\ell_0 + \Delta x)$. This shows that T_i is ndependent of i. Moreover, the extra approximate elongation Δx is very small

compared to $\Delta\ell_0$, so we may well set $T_i = T = k\Delta\ell_0$. This means t tension is completely dominated by the initial tension determined by the of the string. The additional deformations of the spring during the vib do not introduce significant changes in the tension.

19.2 Waves on a membrane

19.3 Elastic waves in a rod

Consider an elastic rod subject to a hammer impact at the end. This exp will give rise to an elastic deformation pulse that travels through the mathematical model for longitudinal waves along an elastic rod starts v general equation for deformations and stresses in an elastic medium,

$$\varrho\boldsymbol{u}_{tt} = \nabla\cdot\boldsymbol{\sigma} + \varrho\boldsymbol{f},$$

where ϱ is the density, $\boldsymbol{u}$ the displacement field, $\boldsymbol{\sigma}$ the stress tensor, and forces. The latter has normally no impact on elastic waves.

For stationary deformation of an elastic rod, one has that $\sigma_{xx} = Eu$ all other stress components being zero. Moreover, $\boldsymbol{u} = u(x)\boldsymbol{i}$. The para is known as Young's modulus. Assuming that this simple stress and defo field, which is exact in the stationary case, is a good approximation transient case with wave motion, (120) simplifies to

$$\varrho\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(E\frac{\partial u}{\partial x}\right)\,.$$

The associated boundary conditions are u or $\sigma_{xx} = Eu_x$ known, t $u = 0$ for a clamped end and $\sigma_{xx} = 0$ for a free end.

19.4 The acoustic model for seismic waves

Seismic waves are used to infer properties of subsurface geological str The physical model is a heterogeneous elastic medium where sound is pro by small elastic vibrations. The general mathematical model for deforma an elastic medium is based on Newton's second law,

$$\varrho\boldsymbol{u}_{tt} = \nabla\cdot\boldsymbol{\sigma} + \varrho\boldsymbol{f},$$

and a constitutive law relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$, often Hooke's generalized law,

$$\boldsymbol{\sigma} = K\nabla\cdot\boldsymbol{u}\,\boldsymbol{I} + G(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T - \frac{2}{3}\nabla\cdot\boldsymbol{u}\,\boldsymbol{I})\,.$$

Here, $\boldsymbol{u}$ is the displacement field, $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{I}$ is the identity t is the medium's density, $\boldsymbol{f}$ are body forces (such as gravity), K is the m bulk modulus and G is the shear modulus. All these quantities may space, while $\boldsymbol{u}$ and $\boldsymbol{\sigma}$ will also show significant variation in time duri motion.

The acoustic approximation to elastic waves arises from a basic assumption ıat the second term in Hooke's law, representing the deformations that give se to shear stresses, can be neglected. This assumption can be interpreted as pproximating the geological medium by a fluid. Neglecting also the body forces , (122) becomes

$$\varrho \boldsymbol{u}_{tt} = \nabla(K\nabla\cdot\boldsymbol{u}) \tag{124}$$

ıtroducing p as a pressure via

$$p = -K\nabla\cdot\boldsymbol{u}, \tag{125}$$

ıd dividing (124) by ϱ, we get

$$\boldsymbol{u}_{tt} = -\frac{1}{\varrho}\nabla p\,. \tag{126}$$

aking the divergence of this equation, using $\nabla\cdot\boldsymbol{u} = -p/K$ from (125), gives ıe *acoustic approximation to elastic waves*:

$$p_{tt} = K\nabla\cdot\left(\frac{1}{\varrho}\nabla p\right)\,. \tag{127}$$

his is a standard, linear wave equation with variable coefficients. It is common ɔ add a source term $s(x,y,z,t)$ to model the generation of sound waves:

$$p_{tt} = K\nabla\cdot\left(\frac{1}{\varrho}\nabla p\right) + s\,. \tag{128}$$

A common additional approximation of (128) is based on using the chain ıle on the right-hand side,

$$K\nabla\cdot\left(\frac{1}{\varrho}\nabla p\right) = \frac{K}{\varrho}\nabla^2 p + K\nabla\left(\frac{1}{\varrho}\right)\cdot\nabla p \approx \frac{K}{\varrho}\nabla^2 p,$$

nder the assumption that the relative spatial gradient $\nabla\varrho^{-1} = -\varrho^{-2}\nabla\varrho$ is small. his approximation results in the simplified equation

$$p_{tt} = \frac{K}{\varrho}\nabla^2 p + s\,. \tag{129}$$

The acoustic approximations to seismic waves are used for sound waves in ıe ground, and the Earth's surface is then a boundary where p equals the tmospheric pressure p_0 such that the boundary condition becomes $p = p_0$.

ınisotropy. Quite often in geological materials, the effective wave velocity $= \sqrt{K/\varrho}$ is different in different spatial directions because geological layers are ɔmpacted such that the properties in the horizontal and vertical direction differ. Vith z as the vertical coordinate, we can introduce a vertical wave velocity c_z ıd a horizontal wave velocity c_h, and generalize (129) to

$$p_{tt} = c_z^2 p_{zz} + c_h^2(p_{xx} + p_{yy}) + s\,. \tag{130}$$

19.5 Sound waves in liquids and gases

Sound waves arise from pressure and density variations in fluids. The : point of modeling sound waves is the basic equations for a compressil where we omit viscous (frictional) forces, body forces (gravity, for instan temperature effects:

$$\begin{aligned}\varrho_t + \nabla\cdot(\varrho\boldsymbol{u}) &= 0,\\ \varrho\boldsymbol{u}_t + \varrho\boldsymbol{u}\cdot\nabla\boldsymbol{u} &= -\nabla p,\\ \varrho &= \varrho(p)\,.\end{aligned}$$

These equations are often referred to as the Euler equations for the mot fluid. The parameters involved are the density ϱ, the velocity $\boldsymbol{u}$, and the p. Equation (132) reflects mass balance, (131) is Newton's second law fo with frictional and body forces omitted, and (133) is a constitutive law density to pressure by thermodynamics considerations. A typical model f is the so-called isentropic relation[30], valid for adiabatic processes where no heat transfer:

$$\varrho = \varrho_0\left(\frac{p}{p_0}\right)^{1/\gamma}\,.$$

Here, p_0 and ϱ_0 are references values for p and ϱ when the fluid is at res is the ratio of specific heat at constant pressure and constant volume (for air).

The key approximation in a mathematical model for sound waves is to that these waves are small perturbations to the density, pressure, and We therefore write

$$\begin{aligned}p &= p_0 + \hat{p},\\ \varrho &= \varrho_0 + \hat{\varrho},\\ \boldsymbol{u} &= \hat{\boldsymbol{u}},\end{aligned}$$

where we have decomposed the fields in a constant equilibrium valu sponding to $\boldsymbol{u} = 0$, and a small perturbation marked with a hat sym inserting these decompositions in (131) and (132), neglecting all produc of small perturbations and/or their derivatives, and dropping the hat s one gets the following linearized PDE system for the small perturba density, pressure, and velocity:

$$\begin{aligned}\varrho_t + \varrho_0\nabla\cdot\boldsymbol{u} &= 0,\\ \varrho_0\boldsymbol{u}_t &= -\nabla p\,.\end{aligned}$$

[30] http://en.wikipedia.org/wiki/Isentropic_process

ow we can eliminate ϱ_t by differentiating the relation $\varrho(p)$,

$$\varrho_t = \varrho_0 \frac{1}{\gamma}\left(\frac{p}{p_0}\right)^{1/\gamma-1}\frac{1}{p_0}p_t = \frac{\varrho_0}{\gamma p_0}\left(\frac{p}{p_0}\right)^{1/\gamma-1}p_t\,.$$

he product term $p^{1/\gamma-1}p_t$ can be linearized as $p_0^{1/\gamma-1}p_t$, resulting in

$$\varrho_t \approx \frac{\varrho_0}{\gamma p_0}p_t\,.$$

/e then get

$$p_t + \gamma p_0 \nabla\cdot\boldsymbol{u} = 0, \tag{137}$$

$$\boldsymbol{u}_t = -\frac{1}{\varrho_0}\nabla p,\,. \tag{138}$$

aking the divergence of (138) and differentiating (137) with respect to time ives the possibility to easily eliminate $\nabla\cdot\boldsymbol{u}_t$ and arrive at a standard, linear ave equation for p:

$$p_{tt} = c^2\nabla^2 p, \tag{139}$$

here $c = \sqrt{\gamma p_0/\varrho_0}$ is the speed of sound in the fluid.

9.6 Spherical waves

pherically symmetric three-dimensional waves propagate in the radial direction only so that $u = u(r,t)$. The fully three-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(c^2\nabla u) + f$$

ıen reduces to the spherically symmetric wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right) + f(r,t), \quad r\in(0,R),\ t>0\,. \tag{140}$$

ne can easily show that the function $v(r,t) = ru(r,t)$ fulfills a standard wave quation in Cartesian coordinates if c is constant. To this end, insert $u = v/r$ in

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right)$$

o obtain

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + c^2\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

he two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(c^2\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one C coordinate. The spherically symmetric wave equation in terms of $v(r$ becomes

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial r}\left(c^2(r)\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{dc^2}{dr}v + rf(r,t), \quad r\in(0,R),\ t>0\,.$$

In the case of constant wave velocity c, this equation reduces to the wave e in a single Cartesian coordinate called r:

$$\frac{\partial^2 v}{\partial t^2} = c^2\frac{\partial^2 v}{\partial r^2} + rf(r,t), \quad r\in(0,R),\ t>0\,.$$

That is, any program for solving the one-dimensional wave equation in a C coordinate system can be used to solve (142), provided the source multiplied by the coordinate, and that we divide the Cartesian mesh sol r to get the spherically symmetric solution. Moreover, if $r = 0$ is include domain, spherical symmetry demands that $\partial u/\partial r = 0$ at $r = 0$, whic that

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r = 0,$$

implying $v(0,t) = 0$ as a necessary condition. For practical applicati exclude $r = 0$ from the domain and assume that some boundary con assigned at $r = \epsilon$, for some $\epsilon > 0$.

19.7 The linear shallow water equations

The next example considers water waves whose wavelengths are much la the depth and whose wave amplitudes are small. This class of waves generated by catastrophic geophysical events, such as earthquakes at bottom, landslides moving into water, or underwater slides (or a comb as earthquakes frequently release avalanches of masses). For example, a earthquake will normally have an extension of many kilometers but lift t only a few meters. The wave length will have a size dictated by the ear area, which is much lager than the water depth, and compared to th length, an amplitude of a few meters is very small. The water is essentiall film, and mathematically we can average the problem in the vertical d and approximate the 3D wave phenomenon by 2D PDEs. Instead of a water domain in three space dimensions, we get a horizontal 2D domain unknown function for the surface elevation and the water depth as a coefficient in the PDEs.

Let $\eta(x,y,t)$ be the elevation of the water surface, $H(x,y)$ the wate corresponding to a flat surface ($\eta = 0$), $u(x,y,t)$ and $v(x,y,t)$ the depth-a horizontal velocities of the water. Mass and momentum balance of th volume give rise to the PDEs involving these quantities:

$$\eta_t = -(Hu)_x - (Hv)_x \tag{143}$$
$$u_t = -g\eta_x, \tag{144}$$
$$v_t = -g\eta_y, \tag{145}$$

here g is the acceleration of gravity. Equation (143) corresponds to mass alance while the other two are derived from momentum balance (Newton's econd law).

The initial conditions associated with (143)-(145) are η, u, and v prescribed t $t = 0$. A common condition is to have some water elevation $\eta = I(x, y)$ and ssume that the surface is at rest: $u = v = 0$. A subsea earthquake usually ıeans a sufficiently rapid motion of the bottom and the water volume to say ıat the bottom deformation is mirrored at the water surface as an initial lift (x, y) and that $u = v = 0$.

Boundary conditions may be η prescribed for incoming, known waves, or ero normal velocity at reflecting boundaries (steep mountains, for instance): $n_x + vn_y = 0$, where (n_x, n_y) is the outward unit normal to the boundary. More ophisticated boundary conditions are needed when waves run up at the shore, nd at open boundaries where we want the waves to leave the computational omain undisturbed.

Equations (143), (144), and (145) can be transformed to a standard, linear ave equation. First, multiply (144) and (145) by H, differentiate (144)) with espect to x and (145) with respect to y. Second, differentiate (143) with espect to t and use that $(Hu)_{xt} = (Hu_t)_x$ and $(Hv)_{yt} = (Hv_t)_y$ when H is ıdependent of t. Third, eliminate $(Hu_t)_x$ and $(Hv_t)_y$ with the aid of the other wo differentiated equations. These manipulations results in a standard, linear ave equation for η:

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla \cdot (gH\nabla\eta)\,. \tag{146}$$

In the case we have an initial non-flat water surface at rest, the initial onditions become $\eta = I(x, y)$ and $\eta_t = 0$. The latter follows from (143) if $= v = 0$, or simply from the fact that the vertical velocity of the surface is η_t, hich is zero for a surface at rest.

The system (143)-(145) can be extended to handle a time-varying bottom opography, which is relevant for modeling long waves generated by underwater ides. In such cases the water depth function H is also a function of t, due to ıe moving slide, and one must add a time-derivative term H_t to the left-hand de of (143). A moving bottom is best described by introducing $z = H_0$ as the ill-water level, $z = B(x, y, t)$ as the time- and space-varying bottom topography, o that $H = H_0 - B(x, y, t)$. In the elimination of u and v one may assume that ıe dependence of H on t can be neglected in the terms $(Hu)_{xt}$ and $(Hv)_{yt}$. We ıen end up with a source term in (146), because of the moving (accelerating) ottom:

$$\eta_{tt} = \nabla \cdot (gH\nabla\eta) + B_{tt}\,. \tag{147}$$

The reduction of (147) to 1D, for long waves in a straight channe approximately plane waves in the ocean, is trivial by assuming no cha direction ($\partial/\partial y = 0$):

$$\eta_t = (gH\eta_x)_x + B_{tt}\,.$$

Wind drag on the surface. Surface waves are influenced by the dra wind, and if the wind velocity some meters above the surface is (U, V), t drag gives contributions $C_V\sqrt{U^2+V^2}U$ and $C_V\sqrt{U^2+V^2}V$ to (144) ar respectively, on the right-hand sides.

Bottom drag. The waves will experience a drag from the botton roughly modeled by a term similar to the wind drag: $C_B\sqrt{u^2+v^2}u$ right-hand side of (144) and $C_B\sqrt{u^2+v^2}v$ on the right-hand side of (14 that in this case the PDEs (144) and (145) become nonlinear and the elin of u and v to arrive at a 2nd-order wave equation for η is not possible a

Effect of the Earth's rotation. Long geophysical waves will often be by the rotation of the Earth because of the Coriolis force. This force g to a term fv on the right-hand side of (144) and $-fu$ on the right-ha of (145). Also in this case one cannot eliminate u and v to work with equation for η. The Coriolis parameter is $f = 2\Omega\sin\phi$, where Ω is the velocity of the earth and ϕ is the latitude.

19.8 Waves in blood vessels

The flow of blood in our bodies is basically fluid flow in a network c Unlike rigid pipes, the walls in the blood vessels are elastic and will i their diameter when the pressure rises. The elastic forces will then push back and accelerate the fluid. This interaction between the flow of blood deformation of the vessel wall results in waves traveling along our blood

A model for one-dimensional waves along blood vessels can be deriv averaging the fluid flow over the cross section of the blood vessels. Le coordinate along the blood vessel and assume that all cross sections are though with different radius $R(x, t)$. The main quantities to comput cross section area $A(x, t)$, the averaged pressure $P(x, t)$, and the total flux $Q(x, t)$. The area of this cross section is

$$A(x,t) = 2\pi\int_0^{R(x,t)} r dr,$$

Let $v_x(x, t)$ be the velocity of blood averaged over the cross section at The volume flux, being the total volume of blood passing a cross sect time unit, becomes

$$Q(x,t) = A(x,t)v_x(x,t)$$

Mass balance and Newton's second law lead to the PDEs

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{151}$$

$$\frac{\partial Q}{\partial t} + \frac{\gamma+2}{\gamma+1}\frac{\partial}{\partial x}\left(\frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} = -2\pi(\gamma+2)\frac{\mu}{\varrho}\frac{Q}{A}, \tag{152}$$

here γ is a parameter related to the velocity profile, ϱ is the density of blood, nd μ is the dynamic viscosity of blood.

We have three unknowns A, Q, and P, and two equations (151) and (152). third equation is needed to relate the flow to the deformations of the wall. A ommon form for this equation is

$$\frac{\partial P}{\partial t} + \frac{1}{C}\frac{\partial Q}{\partial x} = 0, \tag{153}$$

here C is the compliance of the wall, given by the constitutive relation

$$C = \frac{\partial A}{\partial P} + \frac{\partial A}{\partial t}, \tag{154}$$

hich require a relationship between A and P. One common model is to view ıe vessel wall, locally, as a thin elastic tube subject to an internal pressure. his gives the relation

$$P = P_0 + \frac{\pi h E}{(1-\nu^2)A_0}(\sqrt{A} - \sqrt{A_0}),$$

here P_0 and A_0 are corresponding reference values when the wall is not deformed, is the thickness of the wall, and E and ν are Young's modulus and Poisson's ıtio of the elastic material in the wall. The derivative becomes

$$C = \frac{\partial A}{\partial P} = \frac{2(1-\nu^2)A_0}{\pi h E}\sqrt{A_0} + 2\left(\frac{(1-\nu^2)A_0}{\pi h E}\right)^2 (P - P_0)\,. \tag{155}$$

nother (nonlinear) deformation model of the wall, which has a better fit with xperiments, is

$$P = P_0 \exp\left(\beta(A/A_0 - 1)\right),$$

here β is some parameter to be estimated. This law leads to

$$C = \frac{\partial A}{\partial P} = \frac{A_0}{\beta P}\,. \tag{156}$$

eduction to standard wave equation. It is not uncommon to neglect the iscous term on the right-hand side of (152) and also the quadratic term with Q^2 n the left-hand side. The reduced equations (152) and (153) form a first-order near wave equation system:

$$C\frac{\partial P}{\partial t} = -\frac{\partial Q}{\partial x},$$
$$\frac{\partial Q}{\partial t} = -\frac{A}{\varrho}\frac{\partial P}{\partial x}\,.$$

These can be combined into standard 1D wave equation PDE by differe the first equation with respect t and the second with respect to x,

$$\frac{\partial}{\partial t}\left(CC\frac{\partial P}{\partial t}\right) = \frac{\partial}{\partial x}\left(\frac{A}{\varrho}\frac{\partial P}{\partial x}\right),$$

which can be approximated by

$$\frac{\partial^2 Q}{\partial t^2} = c^2\frac{\partial^2 Q}{\partial x^2}, \quad c = \sqrt{\frac{A}{\varrho C}},$$

where the A and C in the expression for c are taken as constant referenc

19.9 Electromagnetic waves

Light and radio waves are governed by standard wave equations arisi Maxwell's general equations. When there are no charges and no curren a vacuum, Maxwell's equations take the form

$$\nabla \cdot \boldsymbol{E} = 0,$$
$$\nabla \cdot \boldsymbol{B} = 0,$$
$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t},$$
$$\nabla \times \boldsymbol{B} = \mu_0\epsilon_0\frac{\partial \boldsymbol{E}}{\partial t},$$

where $\epsilon_0 = 8.854187817620 \cdot 10^{-12}$ (F/m) is the permittivity of free spa known as the electric constant, and $\mu_0 = 1.2566370614 \cdot 10^{-6}$ (H/m permeability of free space, also known as the magnetic constant. Taking of the two last equations and using the identity

$$\nabla \times (\nabla \times \boldsymbol{E}) = \nabla(\nabla \cdot \boldsymbol{E}) - \nabla^2\boldsymbol{E} = -\nabla^2\boldsymbol{E} \text{ when } \nabla \cdot \boldsymbol{E} = 0,$$

immediately gives the wave equation governing the electric and magnet

$$\frac{\partial^2 \boldsymbol{E}}{\partial t^2} = c^2\frac{\partial^2 \boldsymbol{E}}{\partial x^2},$$
$$\frac{\partial^2 \boldsymbol{E}}{\partial t^2} = c^2\frac{\partial^2 \boldsymbol{E}}{\partial x^2},$$

with $c = 1/\sqrt{\mu_0\epsilon_0}$ as the velocity of light. Each component of $\boldsymbol{E}$ and $\boldsymbol{B}$ wave equation and can hence be solved independently.

0 Exercises

Exercise 15: Simulate waves on a non-homogeneous string

imulate waves on a string that consists of two materials with different density. he tension in the string is constant, but the density has a jump at the middle of ıe string. Experiment with different sizes of the jump and produce animations ıat visualize the effect of the jump on the wave motion.

Hint. According to Section 19.1, the density enters the mathematical model as in $\varrho u_{tt} = Tu_{xx}$, where T is the string tension. Modify, e.g., the `wave1D_u0v.py` ode to incorporate the tension and two density values. Make a mesh function `ho` with density values at each spatial mesh point. A value for the tension may e 150 N. Corresponding density values can be computed from the wave velocity stimations in the `guitar` function in the `wave1D_u0v.py` file.
ilename: `wave1D_u0_sv_discont.py`.

Exercise 16: Simulate damped waves on a string

ormulate a mathematical model for damped waves on a string. Use data from ection 3.4, and tune the damping parameter so that the string is very close o the rest state after 15 s. Make a movie of the wave motion. Filename: `ave1D_u0_sv_damping.py`.

Exercise 17: Simulate elastic waves in a rod

hammer hits the end of an elastic rod. The exercise is to simulate the resulting ave motion using the model (121) from Section 19.3. Let the rod have length and let the boundary $x = L$ be stress free so that $\sigma_{xx} = 0$, implying that $u/\partial x = 0$. The left end $x = 0$ is subject to a strong stress pulse (the hammer), ıodeled as

$$\sigma_{xx}(t) = \begin{cases} S, & 0 < t \leq t_s, \\ 0, & t > t_s \end{cases}$$

he corresponding condition on u becomes $u_x = S/E$ for $t \leq t_s$ and zero fterwards (recall that $\sigma_{xx} = Eu_x$). This is a non-homogeneous Neumann ondition, and you will need to approximate this condition and combine it with ıe scheme (the ideas and manipulations follow closely the handling of a non-zero ıitial condition $u_t = V$ in wave PDEs or the corresponding second-order ODEs or vibrations). Filename: `wave_rod.py`.

Exercise 18: Simulate spherical waves

nplement a model for spherically symmetric waves using the method described ı Section 19.6. The boundary condition at $r = 0$ must be $\partial u/\partial r = 0$, while the ondition at $r = R$ can either be $u = 0$ or a radiation condition as described in

Problem 21. The $u = 0$ condition is sufficient if R is so large that the an of the spherical wave has become insignificant. Make movie(s) of the cas the source term is located around $r = 0$ and sends out pulses

$$f(r,t) = \begin{cases} Q \exp\left(-\frac{r^2}{2\Delta r^2}\right) \sin \omega t, & \sin \omega t \geq 0 \\ 0, & \sin \omega t < 0 \end{cases}$$

Here, Q and ω are constants to be chosen.

Hint. Use the program `wave1D_u0v.py` as a starting point. Let `solv` pute the v function and then set $u = v/r$. However, $u = v/r$ for $r = 0$ special treatment. One possibility is to compute `u[1:] = v[1:]/r[1` then set `u[0]=u[1]`. The latter makes it evident that $\partial u/\partial r = 0$ in a p Filename: `wave1D_spherical.py`.

Exercise 19: Explain why numerical noise occurs

The experiments performed in Exercise 8 shows considerable numeric in the form of non-physical waves, especially for $s_f = 4$ and the plu or the half a "cosinehat" pulse. The noise is much less visible for a G pulse. Run the case with the plug and half a "cosinehat" pulses for $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion rel explain the observations. Filename: `pulse1D_analysis.pdf`.

Exercise 20: Investigate harmonic averaging in a 1D ı

Harmonic means are often used if the wave velocity is non-smooth or tinuous. Will harmonic averaging of the wave velocity give less nu noise for the case $s_f = 4$ in Exercise 8? Filenames: `pulse1D_harmon` `pulse1D_harmonic.py`.

Problem 21: Implement open boundary conditions

To enable a wave to leave the computational domain and travel undi through the boundary $x = L$, one can in a one-dimensional problem imp following condition, called a *radiation condition* or *open boundary cond*

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0\,.$$

The parameter c is the wave velocity.

Show that (162) accepts a solution $u = g_R(x - ct)$ (right-going wa not $u = g_L(x + ct)$ (left-going wave). This means that (162) will al right-going wave $g_R(x - ct)$ to pass through the boundary undisturbed

A corresponding open boundary condition for a left-going wave throug is

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0\,. \tag{163}$$

) A natural idea for discretizing the condition (162) at the spatial end point $= N_x$ is to apply centered differences in time and space:

$$[D_{2t}u + cD_{2x}u = 0]_i^n, \quad i = N_x\,. \tag{164}$$

liminate the fictitious value $u^n_{N_x+1}$ by using the discrete equation at the same oint.

The equation for the first step, u_i^1, is in principle also affected, but we can ıen use the condition $u_{N_x} = 0$ since the wave has not yet reached the right oundary.

) A much more convenient implementation of the open boundary condition at $= L$ can be based on an explicit discretization

$$[D_t^+u + cD_x^-u = 0]_i^n, \quad i = N_x\,. \tag{165}$$

rom this equation, one can solve for $u^{n+1}_{N_x}$ and apply the formula as a Dirichlet ɔndition at the boundary point. However, the finite difference approximations ıvolved are of first order.

Implement this scheme for a wave equation $u_{tt} = c^2u_{xx}$ in a domain $[0, L]$, here you have $u_x = 0$ at $x = 0$, the condition (162) at $x = L$, and an initial isturbance in the middle of the domain, e.g., a plug profile like

$$u(x,0) = \begin{cases} 1, & L/2 - \ell \leq x \leq L/2 + \ell, \\ 0, \text{otherwise} \end{cases}$$

)bserve that the initial wave is split in two, the left-going wave is reflected at $= 0$, and both waves travel out of $x = L$, leaving the solution as $u = 0$ in $[0, L]$. se a unit Courant number such that the numerical solution is exact. Make a ıovie to illustrate what happens.

Because this simplified implementation of the open boundary condition works, ıere is no need to pursue the more complicated discretization in a).

Hint. Modify the solver function in `wave1D_dn.py`[31].

) Add the possibility to have either $u_x = 0$ or an open boundary condition at ıe left boundary. The latter condition is discretized as

$$[D_t^+u - cD_x^+u = 0]_i^n, \quad i = 0, \tag{166}$$

ading to an explicit update of the boundary value u_0^{n+1}.

The implementation can be tested with a Gaussian function as initial condi- on:

[31] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn.py`

$$g(x; m, s) = \frac{1}{\sqrt{2\pi}s}e^{-\frac{(x-m)^2}{2s^2}}\,.$$

Run two tests:

1. Disturbance in the middle of the domain, $I(x) = g(x; L/2, s)$, aı boundary condition at the left end.
2. Disturbance at the left end, $I(x) = g(x; 0, s)$, and $u_x = 0$ as sy boundary condition at this end.

Make nose tests for both cases, testing that the solution is zero after th have left the domain.

d) In 2D and 3D it is difficult to compute the correct wave velocity nc the boundary, which is needed in generalizations of the open boundary co in higher dimensions. Test the effect of having a slightly wrong wave vel (165). Make a movies to illustrate what happens.
Filename: `wave1D_open_BC.py`.

Remarks. The condition (162) works perfectly in 1D when c is knowı and 3D, however, the condition reads $u_t + c_xu_x + c_yu_y = 0$, where c_x are the wave speeds in the x and y directions. Estimating these com (i.e., the direction of the wave) is often challenging. Other methods are ı used in 2D and 3D to let waves move out of a computational domain.

Exercise 22: Implement periodic boundary condition

It is frequently of interest to follow wave motion over large distances a times. A straightforward approach is to work with a very large dom might lead to a lot of computations in areas of the domain where th cannot be noticed. A more efficient approach is to let a right-going w of the domain and at the same time let it enter the domain on the left. called a *periodic boundary condition*.

The boundary condition at the right end $x = L$ is an open boundary c (see Exercise 21) to let a right-going wave out of the domain. At the $x = 0$, we apply, in the beginning of the simulation, either a symmetry b condition (see Exercise 7) $u_x = 0$, or an open boundary condition.

This initial wave will split in two and either reflected or transporte the domain at $x = 0$. The purpose of the exercise is to follow the rigl wave. We can do that with a *periodic boundary condition*. This means th the right-going wave hits the boundary $x = L$, the open boundary condi the wave out of the domain, but at the same time we use a boundary c on the left end $x = 0$ that feeds the outgoing wave into the domain aga periodic condition is simply $u(0) = u(L)$. The switch from $u_x = 0$ or a boundary condition at the left end to a periodic condition can happe

$(L,t) > \epsilon$, where $\epsilon = 10^{-4}$ might be an appropriate value for determining when ıe right-going wave hits the boundary $x = L$.

The open boundary conditions can conveniently be discretized as explained ı Exercise 21. Implement the described type of boundary conditions and ɔst them on two different initial shapes: a plug $u(x,0) = 1$ for $x \leq 0.1$, $(x,0) = 0$ for $x > 0.1$, and a Gaussian function in the middle of the domain: $(x,0) = \exp(-\frac{1}{2}(x-0.5)^2/0.05)$. The domain is the unit interval $[0,1]$. Run ıese two shapes for Courant numbers 1 and 0.5. Assume constant wave velocity. Iake movies of the four cases. Reason why the solutions are correct. Filename: `eriodic.py`.

'roblem 23: Earthquake-generated tsunami over a subsea .ill

subsea earthquake leads to an immediate lift of the water surface, see Figure 10. he lifted water surface splits into two tsunamis, one traveling to the right and ne to the left, as depicted in Figure 11. Since tsunamis are normally very long aves, compared to the depth, with a small amplitude, compared to the wave ngth, the wave equation model described in Section 19.7 is relevant:

$$\eta_{tt} = (gH(x)\eta_x)_x,$$

here g is the acceleration of gravity, and $H(x)$ is the still water depth.

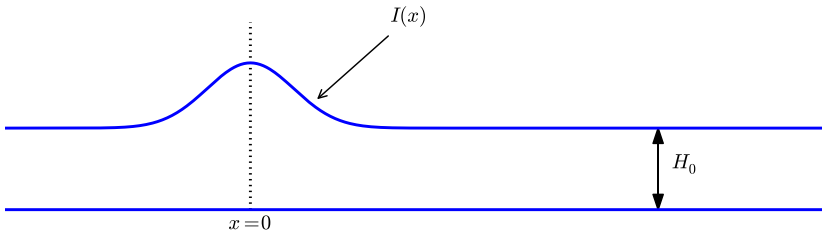


Figure 10: Sketch of initial water surface due to a subsea earthquake.

To simulate the right-going tsunami, we can impose a symmetry boundary t $x = 0$: $\partial\eta\,\partial x = 0$. We then simulate the wave motion in $[0,L]$. Unless the ɔean ends at $x = L$, the waves should travel undisturbed through the boundary $= L$. A radiation condition as explained in Problem 21 can be used for this urpose. Alternatively, one can just stop the simulations before the wave hits

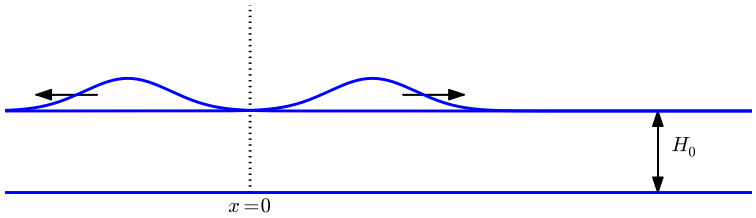


Figure 11: An initial surface elevation is split into two waves.

the boundary at $x = L$. In that case it does not matter what kind of bc condition we use at $x = L$. Imposing $\eta = 0$ and stopping the simulatioı $|\eta_i^n| > \epsilon$, $i = N_x - 1$, is a possibility (ϵ is a small parameter).

The shape of the initial surface can be taken as a Gaussian functioı

$$I(x; I_0, I_a, I_m, I_s) = I_0 + I_a \exp\left(-\left(\frac{x - I_m}{I_s}\right)^2\right),$$

with $I_m = 0$ reflecting the location of the peak of $I(x)$ and I_s being a ı of the width of the function $I(x)$ (I_s is $\sqrt{2}$ times the standard deviatio familiar normal distribution curve).

Now we extend the problem with a hill at the sea bottom, see Figure wave speed $c = \sqrt{gH(x)} = \sqrt{g(H_0 - B(x))}$ will then be reduced in the water above the hill.

One possible form of the hill is a Gaussian function,

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \exp\left(-\left(\frac{x - B_m}{B_s}\right)^2\right),$$

but many other shapes are also possible, e.g., a "cosine hat" where

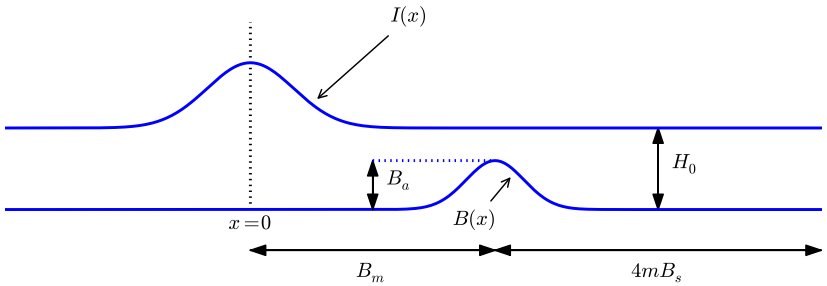
$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \cos\left(\pi\frac{x - B_m}{2B_s}\right),$$

when $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

Also an abrupt construction may be tried:

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a,$$

for $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.



igure 12: Sketch of an earthquake-generated tsunami passing over a subsea ill.

The `wave1D_dn_vc.py`[32] program can be used as starting point for the nplementation. Visualize both the bottom topography and the water surface levation in the same plot. Allow for a flexible choice of bottom shape: (168), .69), (170), or $B(x) = B_0$ (flat).

The purpose of this problem is to explore the quality of the numerical solution $\frac{n}{i}$ for different shapes of the bottom obstruction. The "cosine hat" and the box-haped hills have abrupt changes in the derivative of $H(x)$ and are more likely to enerate numerical noise than the smooth Gaussian shape of the hill. Investigate this is true. Filenames: `tsunami1D_hill.py`, `tsunami1D_hill.pdf`.

roblem 24: Earthquake-generated tsunami over a 3D hill

his problem extends Problem 23 to a three-dimensional wave phenomenon, overned by the 2D PDE (146). We assume that the earthquake arise from a ult along the line $x = 0$ in the xy-plane so that the initial lift of the surface an be taken as $I(x)$ in Problem 23. That is, a plane wave is propagating to the ght, but will experience bending because of the bottom.

The bottom shape is now a function of x and y. An "elliptic" Gaussian nction in two dimensions, with its peak at (B_{mx}, B_{my}), generalizes (168):

$$(x; B_0, B_a, B_{mx}, B_{my}, B_s, b) = B_0 + B_a \exp\left(-\left(\frac{x - B_{mx}}{B_s}\right)^2 - \left(\frac{y - B_{my}}{bB_s}\right)^2\right), \tag{171}$$

[32] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

where b is a scaling parameter: $b = 1$ gives a circular Gaussian functi circular contour lines, while $b \neq 1$ gives an elliptic shape with elliptic lines.

The "cosine hat" (169) can also be generalized to

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_{mx}}{2B_s}\right) \cos\left(\pi \frac{y - B}{2B}\right.$$

when $0 \leq \sqrt{x^2 + y^2} \leq B_s$ and $B = B_0$ outside this circle.

A box-shaped obstacle means that

$$B(x; B_0, B_a, B_m, B_s, b) = B_0 + B_a$$

for x and y inside a rectangle

$$B_{mx} - B_s \leq x \leq B_{mx} + B_s, \quad B_{my} - bB_s \leq y \leq B_{my} + bB_s,$$

and $B = B_0$ outside this rectangle. The b parameter controls the rect shape of the cross section of the box.

Note that the initial condition and the listed bottom shapes are syı around the line $y = B_{my}$. We therefore expect the surface elevation be symmetric with respect to this line. This means that we can ha computational domain by working with $[0, L_x] \times [0, B_{my}]$. Along th boundary, $y = B_{my}$, we must impose the symmetry condition $\partial\eta/\partial n =$ a symmetry condition ($-\eta_x = 0$) is also needed at the $x = 0$ boundary the initial condition has a symmetry here. At the lower boundary $y = 0$ set a Neumann condition (which becomes $-\eta_y = 0$). The wave motion simulated until the wave hits the reflecting boundaries where $\partial\eta/\partial n =$ (one can also set $\eta = 0$ - the particular condition does not matter as lon simulation is stopped before the wave is influenced by the boundary co

Visualize the surface elevation. Investigate how different hill sha ferent sizes of the water gap above the hill, and different resolution $\Delta y = h$ and Δt influence the numerical quality of the solution. Fil `tsunami2D_hill.py`, `tsunami2D_hill.pdf`.

Problem 25: Investigate Matplotlib for visualization

Play with native Matplotlib code for visualizing 2D solutions of the wave e with variable wave velocity. See if there are effective ways to visualize b solution and the wave velocity. Filename: `tsunami2D_hill_mpl.py`.

Problem 26: Investigate visualization packages

Create some fancy 3D visualization of the water waves *and* the subse Problem 24. Try to make the hill transparent. Possible visualization to

- Mayavi[33]
- Paraview[34]
- OpenDX[35]

ilename: `tsunami2D_hill_viz.py`.

roblem 27: Implement loops in compiled languages

xtend the program from Problem 24 such that the loops over mesh points, inside ıe time loop, are implemented in compiled languages. Consider implementations ı Cython, Fortran via `f2py`, C via Cython, C via `f2py`, C/C++ via Instant, nd C/C++ via `scipy.weave`. Perform efficiency experiments to investigate the elative performance of the various implementations. It is often advantageous ɔ normalize CPU times by the fastest method on a given mesh. Filename: `sunami2D_hill_compiled.py`.

xercise 28: Simulate seismic waves in 2D

he goal of this exercise is to simulate seismic waves using the PDE model .30) in a 2D xz domain with geological layers. Introduce m horizontal layers f thickness h_i, $i = 0, \ldots, m-1$. Inside layer number i we have a vertical wave elocity $c_{z,i}$ and a horizontal wave velocity $c_{h,i}$. Make a program for simulating ıch 2D waves. Test it on a case with 3 layers where

$$c_{z,0} = c_{z,1} = c_{z,2}, \quad c_{h,0} = c_{h,2}, \quad c_{h,1} \ll c_{h,0}\,.$$

et s be a localized point source at the middle of the Earth's surface (the pper boundary) and investigate how the resulting wave travels through the ıedium. The source can be a localized Gaussian peak that oscillates in time ɔr some time interval. Place the boundaries far enough from the expanding ave so that the boundary conditions do not disturb the wave. Then the type f boundary condition does not matter, except that we physically need to have $= p_0$, where p_0 is the atmospheric pressure, at the upper boundary. Filename: `eismic2D.py`.

roject 29: Model 3D acoustic waves in a room

he equation for sound waves in air is derived in Section 19.5 and reads

$$p_{tt} = c^2\nabla^2 p,$$

here $p(x,y,z,t)$ is the pressure and c is the speed of sound, taken as 340 m/s. owever, sound is absorbed in the air due to relaxation of molecules in the gas.

[33] `http://code.enthought.com/projects/mayavi/`
[34] `http://www.paraview.org/`
[35] `http://www.opendx.org/`

A model for simple relaxation, valid for gases consisting only of one molecules, is a term $c^2\tau_s\nabla^2 p_t$ in the PDE, where τ_s is the relaxation tim generate sound from, e.g., a loudspeaker in the room, this sound sour also be added to the governing equation.

The PDE with the mentioned type of damping and source then bec

$$p_t t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f,$$

where $f(x,y,z,t)$ is the source term.

The walls can absorb some sound. A possible model is to have a "wa (thicker than the physical wall) outside the room where c is changed su some of the wave energy is reflected and some is absorbed in the wa absorption of energy can be taken care of by adding a damping term bp_t equation:

$$p_t t + bp_t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f\,.$$

Typically, $b = 0$ in the room and $b > 0$ in the wall. A discontinuity i will give rise to reflections. It can be wise to use a constant c in the control reflections because of the discontinuity between c in the air an wall, while b is gradually increased as we go into the wall to avoid ref because of rapid changes in b. At the outer boundary of the wall the co $p = 0$ or $\partial p/\partial n = 0$ can be imposed. The waves should anyway be approx dampened to $p = 0$ this far out in the wall layer.

There are two strategies for discretizing the $\nabla^2 p_t$ term: using a difference between times $n+1$ and $n-1$ (if the equation is sampled at or use a one-sided difference based on levels n and $n-1$. The latter advantage of not leading to any equation system, while the former is secon accurate as the scheme for the simple wave equation $p_t t = c^2\nabla^2 p$. To a equation system, go for the one-sided difference such that the overall becomes explicit and only of first order in time.

Develop a 3D solver for the specified PDE and introduce a wall layer. solver with the method of manufactured solutions. Make some demons where the wall reflects and absorbs the waves (reflection because of disco in b and absorption because of growing b). Experiment with the impac τ_s parameter. Filename: `acoustics.py`.

Project 30: Solve a 1D transport equation

We shall study the wave equation

$$u_t + cu_x = 0, \quad x \in (0,L],\ t \in (0,T],$$

with initial condition

$$u(x,0) = I(x), \quad x \in [0,L],$$

and *one* periodic boundary condition

$$u(0,t) = u(L,t)\,. \tag{178}$$

his boundary condition means that what goes out of the domain at $x = L$ omes in at $x = 0$. Roughly speaking, we need only one boundary condition ecause of the spatial derivative is of first order only.

hysical interpretation. The parameter c can be constant or variable, $c = (x)$. The equation (176) arises in *transport* problems where a quantity u, which ould be temperature or concentration of some contaminant, is transported with ie velocity c of a fluid. In addition to the transport imposed by "travelling with ie fluid", u may also be transported by diffusion (such as heat conduction or ickian diffusion), but we have in the model $u_t + cu_x$ assumed that diffusion ffects are negligible, which they often are.

A widely used numerical scheme for (176) applies a forward difference in me and a backward difference in space when $c > 0$:

$$[D_t^+ u + cD_x^- u = 0]_i^n\,. \tag{179}$$

or $c < 0$ we use a forward difference in space: $[cD_x^+ u]_i^n$.

We shall hereafter assume that $= c(x) > 0$.

To compute (184) we need to integrate $1/c$ to obtain C and then compute ie inverse of C.

The inverse function computation can be easily done if we first think discretely. ay we have some function $y = g(x)$ and seeks its inverse. Plotting (x_i, y_i), here $y_i = g(x_i)$ for some mesh points x_i, displays g as a function of x. The iverse function is simply x as a function of g, i.e., the curve with points (y_i, x_i). Ve can therefore quickly compute points at the curve of the inverse function. ne way of extending these points to a continuous function is to assume a linear ariation (known as linear interpolation) between the points (which actually ieans to draw straight lines between the points, exactly as done by a plotting rogram).

The function `wrap2callable` in `scitools.std` can take a set of points and eturn a continuous function that corresponds to linear variation between the oints. The computation of the inverse of a function g on $[0, L]$ can then be one by

```
def inverse(g, domain, resolution=101):
    x = linspace(domain[0], domain[L], resolution)
    y = g(x)
    from scitools.std import wrap2callable
    g_inverse = wrap2callable((y, x))
    return g_inverse
```

To compute $C(x)$ we need to integrate $1/c$, which can be done by a Trapezoidal ule. Suppose we have computed $C(x_i)$ and need to compute $C(x_{i+1})$. Using ie Trapezoidal rule with m subintervals over the integration domain $[x_i, x_{i+1}]$ ives

$$C(x_{i+1}) = C(x_i) + \int_{x_i}^{x_{i+1}} \frac{dx}{c} \approx h\left(\frac{1}{2}\frac{1}{c(x_i)} + \frac{1}{2}\frac{1}{c(x_{i+1})} + \sum_{j=1}^{m-1} \frac{1}{c(x_i + $$

where $h = (x_{i+1} - x_i)/m$ is the length of the subintervals used for the over $[x_i, x_{i+1}]$. We observe that (180) is a *difference equation* which we c by repeatedly applying (180) for $i = 0, 1, \ldots, N_x - 1$ if a mesh $x_0, x, \ldots$ prescribed. Note that $C(0) = 0$.

a) Show that under the assumption of $a = \text{const}$,

$$u(x,t) = I(x - ct)$$

fulfills the PDE as well as the initial and boundary condition (provided $I(L)$).

b) Set up a computational algorithm and implement it in a function. A is constant and positive.

c) Test implementation by using the remarkable property that the nu solution is exact at the mesh points if $\Delta t = c^{-1}\Delta x$.

d) Make a movie comparing the numerical and exact solution for the f two choices of initial conditions:

$$I(x) = \left[\sin\left(\pi\frac{x}{L}\right)\right]^{2n}$$

where n is an integer, typically $n = 5$, and

$$I(x) = \exp\left(-\frac{(x - L/2)^2}{2\sigma 2}\right).$$

Choose $\Delta t = c^{-1}\Delta x, 0.9c^{-1}\Delta x, 0.5c^{-1}\Delta x$.

e) The performance of the suggested numerical scheme can be inve by analyzing the numerical dispersion relation. Analytically, we have t *Fourier component*

$$u(x,t) = e^{i(kx - \omega t)},$$

is a solution of the PDE if $\omega = kc$. This is the *analytical dispersion rel* complete solution of the PDE can be built by adding up such Fourier com with different amplitudes, where the initial condition I determines the am The solution u is then represented by a Fourier series.

A similar discrete Fourier component at (x_p, t_n) is

$$u_p^q = e^{i(kp\Delta x - \tilde{\omega} n\Delta t)},$$

here in general $\tilde{\omega}$ is a function of k, Δt, and Δx, and differs from the exact $= kc$.

Insert the discrete Fourier component in the numerical scheme and derive an xpression for $\tilde{\omega}$, i.e., the discrete dispersion relation. Show in particular that the $\Delta t/(c\Delta x) = 1$, the discrete solution coincides with the exact solution at ıe mesh points, regardless of the mesh resolution (!). Show that if the stability ɔndition

$$\frac{\Delta t}{c\Delta x} \leq 1,$$

ıe discrete Fourier component cannot grow (i.e., $\tilde{\omega}$ is real).

) Write a test for your implementation where you try to use information from ıe numerical dispersion relation.

) Set up a computational algorithm for the variable coefficient case and im-lement it in a function. Make a test that the function works for constant .

) It can be shown that for an observer moving with velocity $c(x)$, u is constant. his can be used to derive an exact solution when a varies with x. Show first ıat

$$u(x,t) = f(C(x) - t), \tag{184}$$

here

$$C'(x) = \frac{1}{c(x)},$$

a solution of (176) for any differentiable function f.

) Use the initial condition to show that an exact solution is

$$u(x,t) = I(C^{-1}(C(x) - t)),$$

ith C^{-1} being the inverse function of $C = \int c^1 dx$. Since $C(x)$ is an integral $_0^x(1/c)dx$, $C(x)$ is monotonically increasing and there exists hence an inverse ınction C^{-1} with values in $[0, L]$.

) Implement a function for computing $C(x_i)$ and one for computing $C^{-1}(x)$ for ny x. Use these two functions for computing the exact solution $I(C^{-1}(C(x)-t))$. nd up with a function `u_exact_variable_c(x, n, c, I)` that returns the alue of $I(C^{-1}(C(x) - t_n))$.

k) Make movies showing a comparison of the numerical and exact solut the two initial conditions (182) and (30). Choose $\Delta t = \Delta x/\max_{0,L} c(x)$ velocity of the medium as

1. $c(x) = 1 + \epsilon \sin(k\pi x/L)$, $\epsilon < 1$,
2. $c(x) = 1 + I(x)$, where I is given by (182) or (30).

The PDE $u_t + cu_x = 0$ expresses that the initial condition $I(x)$ is tran with velocity $c(x)$.
Filename: `advec1D.py`.

Problem 31: General analytical solution of a 1D da wave equation

We consider an initial-boundary value problem for the damped wave ec

$$\begin{aligned} u_{tt} + bu_t &= c^2 u_{xx}, && x \in (0,L),\ t \in (0,T] \\ u(0,t) &= 0, \\ u(L,t) &= 0, \\ u(x,0) &= I(x), \\ u_t(x,0) &= V(x)\,. \end{aligned}$$

Here, $b \geq 0$ and c are given constants. The aim is to derive a general ar solution of this problem. Familiarity with the method of separation of v for solving PDEs will be assumed.

a) Seek a solution on the form $u(x,t) = X(x)T(t)$. Insert this solutio PDE and show that it leads to two differential equations for X and T:

$$T'' + bT' + \lambda T = 0, \quad c^2 X'' + \lambda X = 0,$$

with $X(0) = X(L) = 0$ as boundary conditions, and λ as a constar determined.

b) Show that $X(x)$ is on the form

$$X_n(x) = C_n \sin kx, \quad k = \frac{n\pi}{L}, \quad n = 1, 2, \ldots$$

where C_n is an arbitrary constant.

c) Under the assumption that $(b/2)^2 < k^2$, show that $T(t)$ is on the fo

$$T_n(t) = e^{-\frac{1}{2}bt}(a_n \cos \omega t + b_n \sin \omega t), \quad \omega = \sqrt{k^2 - \frac{1}{4}b^2}, \quad n = 1, 2,$$

The complete solution is then

$$u(x,t) = \sum_{n=1}^{\infty} \sin kx e^{-\frac{1}{2}bt}(A_n \cos \omega t + B_n \sin \omega t),$$

here the constants A_n and B_n must be computed from the initial conditions.

) Derive a formula for A_n from $u(x,0) = I(x)$ and developing $I(x)$ as a sine ourier series on $[0, L]$.

) Derive a formula for B_n from $u_t(x,0) = V(x)$ and developing $V(x)$ as a sine ourier series on $[0, L]$.

) Calculate A_n and B_n from vibrations of a string where $V(x) = 0$ and

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{185}$$

) Implement the series for $u(x,t)$ in a function `u_series(x, t, tol=1E-10)`, here `tol` is a tolerance for truncating the series. Simply sum the terms until $|a_n|$ and $|b_b|$ both are less than `tol`.

) What will change in the derivation of the analytical solution if we have $_x(0,t) = u_x(L,t) = 0$ as boundary conditions? And how will you solve the roblem with $u(0,t) = 0$ and $u_x(L,t) = 0$?
ilename: `damped_wave1D.pdf`.

roblem 32: General analytical solution of a 2D damped ave equation

arry out Problem 31 in the 2D case: $u_{tt} + bu_t = c^2(u_{xx} + u_{yy})$, where $(x,y) \in), L_x) \times (0, L_y)$. Assume a solution on the form $u(x,y,t) = X(x)Y(y)T(t)$.
ilename: `damped_wave2D.pdf`.

Index

arithmetic mean, 37
array slices, 21
averaging
  arithmetic, 37
  geometric, 37
  harmonic, 37

boundary condition
  open (radiation), 102
boundary conditions
  Dirichlet, 29
  Neumann, 29
  periodic, 104

C extension module, 75
C/Python array storage, 80
column-major ordering, 80
Courant number, 55
Cython, 72
`cython -a` (Python-C translation in HTML), 75

declaration of variables in Cython, 73
Dirichlet conditions, 29
discrete Fourier transform, 52
`distutils`, 75

Fortran array storage, 80
Fortran subroutine, 77
Fourier series, 52
Fourier transform, 52

geometric mean, 37

harmonic average, 37
homogeneous Dirichlet conditions, 29
homogeneous Neumann conditions, 29

index set notation, 31, 68

lambda function (Python), 24

mesh
  finite differences, 4
mesh function, 5

Neumann conditions, 29
`nose` tests, 15

open boundary condition, 102

periodic boundary conditions,

radiation condition, 102
row-major ordering, 80

scalar code, 21
`setup.py`, 75
slice, 21
software testing
  nose, 15
stability criterion, 55
stencil
  1D wave equation, 5
  Neumann boundary, 30

unit testing, 15

vectorization, 21

wave equation
  1D, 3
  1D, analytical properties,
  1D, exact numerical soluti
  1D, finite difference meth
  1D, implementation, 14
  1D, stability, 55
  2D, implementation, 66
waves
  on a string, 3
wrapper code, 77